Übersetzungsmethoden für strukturprogrammierbare Rechner

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der Fakultät für Informatik

der Universität Karlsruhe (Technische Hochschule)

genehmigte

Dissertation

von

Markus Weinhardt

aus Tuttlingen

Tag der mündlichen Prüfung:	1. Juli 1997
Erster Gutachter:	Prof. Dr. Gerhard Goos
Zweiter Gutachter:	Prof. Dr. Detlef Schmid

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Weinhardt, Markus:
Übersetzungsmethoden für strukturprogrammierbare Rechner /
von Markus Weinhardt. - Berlin : Logos-Verl., 1997

Zugl.: Karlsruhe, Univ., Diss., 1997

ISBN 3-89722-011-3

ISBN 3-89722-011-3

Logos Verlag Berlin
Hufelandstr. 34
10407 Berlin
Tel.: 030 - 42851090
INTERNET: http://www.logos-verlag.de/

Vorwort

Diese Arbeit wurde durch ein Promotionsstipendium im Graduiertenkolleg "Beherrschbarkeit komplexer Systeme" der Fakultät für Informatik an der Universität Karlsruhe ermöglicht.

Zunächst möchte ich Herrn Professor G. Goos für die Übernahme des Hauptreferates danken. Er hat mich auf das interessante, neue Forschungsgebiet der rekonfigurierbaren Hardware aufmerksam gemacht und durch wertvolle Hinweise wesentlich zum Gelingen dieser Arbeit beigetragen.

Desweiteren bin ich Herrn Professor D. Schmid für die Übernahme des Koreferates zu Dank verpflichtet. Seine Kommentare und Anregungen haben zur Abrundung und Präzisierung der Arbeit geführt.

Außerdem war die Kritik und Ermutigung durch die KollegInnen am Lehrstuhl -- auch in schwierigen Phasen -- unverzichtbar. Stellvertretend für alle sei an dieser Stelle meinen "Büronachbarn" Horst Moldenhauer und Sabine Glesner gedankt.

Schließlich möchte ich mich bei meinen Eltern und bei meinen Freunden für die Unterstützung und Ermunterung während meiner Studien- und Promotionszeit bedanken.

Markus Weinhardt

Inhaltsverzeichnis

Kapitel 1

Einleitung

Vor etwa einem Jahrzehnt wurde eine neue Klasse programmierbarer Logikbausteine eingeführt. Diese *Field-Programmable Gate Array (FPGA)* genannten Chips werden vom Anwender konfiguriert und können beliebige digitale Schaltungen realisieren. Anfangs wurden FPGAs vor allem zum Testen neuer Schaltungsentwürfe (Rapid Prototyping), zur Realisierung unstrukturierter Logik und für Produkte mit kleinen Stückzahlen, bei denen sich die Produktion eines kundenspezifischen Schaltkreises nicht lohnt, verwendet.

Mit der Entwicklung größerer, mehrfach programmierbarer (rekonfigurierbarer) FPGAs wurde auch der Einsatz als eigenständiger, flexibler Prozessor zunehmend interessant. Mit den heutigen FPGAs, deren Komplexität bis zu 100.000 logischen Gattern entspricht, können auch kompliziertere arithmetisch-logische Funktionseinheiten realisiert werden. Rekonfigurierbare FPGAs vereinen damit die Flexibilität von Software mit der Leistungsfähigkeit anwendungsspezifischer Hardware.

Vielversprechende Einsatzmöglichkeiten bietet die Kombination eines konventionellen, mikroprozessorbasierten Rechners (Wirt) mit einem rekonfigurierbaren Koprozessor, der aus einem oder mehreren FPGAs und lokalem Speicher besteht. Die resultierende Architektur, die wir *strukturprogrammierbaren Rechner* nennen, kann eine Anwendung teilweise auf dem Mikroprozessor des Wirts ausführen und teilweise auf den Koprozessor auslagern. Jedoch ist eine solche Anwendung nur schwer zu erstellen, da neben Erfahrung in der Software-Entwicklung auch Kenntnisse im Hardware-Entwurf erforderlich sind. Außerdem muß die Schnittstelle zwischen beiden Teilen manuell implementiert werden.

Für einen breiteren Einsatz dieser Rechner — auch durch Software-Entwickler — ist es deshalb vorteilhaft, die gesamte Anwendung in einer höheren Programmiersprache zu spezifizieren. Zur Weiterverarbeitung wird dann jedoch ein Übersetzer benötigt, der aus einem Eingabeprogramm automatisch eine Anwendung für den strukturprogrammierbaren Rechner generiert. In dieser Arbeit werden Methoden für einen solchen kombinierten Übersetzer entwickelt.

Dabei müssen folgende Teilprobleme gelöst werden: Zunächst muß die *Kandidaten*

auswahl diejenigen Programmteile bestimmen, die überhaupt durch rekonfigurierbare Hardware ausgeführt werden können. Im nächsten Schritt, der *Koprozessor-Synthese*, müssen für die Kandidaten aus der Programmdarstellung Hardware-Beschreibungen generiert werden. Schließlich erfolgt die *Integration* der Hardware- und Software-Anteile. Dabei muß auch eine *Hardware/Software-Partitionierung* berücksichtigen, wann und für welche Kandidaten sich die Verlagerung in Hardware überhaupt lohnt.

Die in dieser Arbeit entwickelten Methoden zeigen Lösungsmöglichkeiten für die genannten Probleme auf: Der Kern der Arbeit ist ein neues Verfahren, das erstmals aus einer imperativen, sequentiellen Eingabesprache strukturelle Schaltungsbeschreibungen erzeugt, welche die Parallelität der rekonfigurierbaren Hardware gut ausnutzen. Die Schaltungen beruhen auf dem im Hardware-Entwurf seit langem erfolgreich eingesetzten Pipeline-Prinzip, wodurch ein hoher Durchsatz ermöglicht wird. Für diese Koprozessor-Synthese wurde ein Analyseverfahren entwickelt, das die von Übersetzern für Vektorrechner bekannte Schleifen-Vektorisierung für die wesentlich flexiblere rekonfigurierbare Hardware erweitert. Da nach der Vektorisierung die in einer Schleife vorhandenen Abhängigkeiten genau bekannt sind, können effiziente Pipeline-Schaltungen synthetisiert werden, deren Parallelität nur durch diese Abhängigkeiten begrenzt wird. Als Hardware-Kandidaten werden deshalb die vektorisierbaren Schleifen ausgewählt.

Um die verfügbaren FPGAs mit diesen Pipelines optimal auszunutzen, wurde ein globales Optimierungsverfahren entwickelt, das die Register in einem Datenfluß-Graphen so verteilt, daß mit einer minimalen Anzahl verwendeter FPGA-Flipflops der maximale Durchsatz erreicht wird. So wird das Zeitverhalten und der Platzbedarf der erzeugten Schaltungen optimiert.

Weiterhin wurde zur automatischen Hardware/Software-Integration eine allgemeine Schnittstelle definiert, die maschinenspezifische Details in begrenzten Programm- bzw. Schaltungsteilen kapselt. So ist die Pipeline-Synthese maschinenunabhängig, während die genaue Architektur des Wirts und der Koprozessor-Karte nur für die einmalige Entwicklung von Steuer-Funktionen und -Modulen, die von allen Programmen gleichermaßen verwendet werden, relevant ist. Die Funktionen werden in das auf dem Wirt laufende Programm eingefügt und die Module in die Koprozessor-Schaltung integriert, so daß die Pipelines automatisch aufgerufen werden können. Dieses Schnittstellen-Konzept wurde für eine Sun-Workstation mit einer kleinen FPGA-Karte implementiert und zusammen mit der Koprozessor-Synthese in den bestehenden Modula-2-Übersetzer MOCKA integriert, um die praktische Anwendbarkeit der Verfahren zu demonstrieren.

Schließlich wurde ein Partitionierungsverfahren entwickelt, das für die Kandidaten abschätzt, ob sich die Auslagerung auf einen Koprozessor überhaupt lohnt. Denn in einigen Fällen wird die Beschleunigung durch den zusätzlichen Zeitbedarf für die Konfigurierung der FPGAs und für die Datenübertragung zwischen Wirt und Koprozessor zunichte gemacht. Das Verfahren beruht auf einem allgemeinen Maschinenmodell, das von den speziellen Eigenschaften eines Rechners abstrahiert. Im

Gegensatz zu aus dem Hardware/Software-Codesign bekannten Methoden verwendet die automatische Auswahl nicht nur Informationen, die durch Profiling oder durch eine statische Analyse während der Übersetzung gewonnen werden. Vielmehr geht auch die aktuelle Größe der Anwendungsdaten während des Programmlaufs in die Entscheidung ein. Außerdem wird automatisch bestimmt, ob sich die Umkonfigurierung des Koprozessors — auch zwischen verschiedenen Phasen eines Programms — lohnt.

Der Rest dieser Arbeit gliedert sich folgendermaßen: Im nächsten Kapitel wird ein erstes Beispiel für den Einsatz eines strukturprogrammierbaren Rechners vorgestellt und ein gröber Überblick über einen kombinierten Übersetzer gegeben. Danach stellt Kapitel 3 die Grundlagen strukturprogrammierbarer Hardware und Kapitel 4 die für diese Arbeit relevanten verwandten Techniken vor.

Die Kapitel 5 und 6 stellen den Kern der Arbeit dar: Kapitel 5 führt die auf Vektorisierung und Pipeline-Synthese beruhende neue Methode zur Generierung rekonfigurierbarer Koprozessoren ein. Danach beschreibt Kapitel 6 die kombinierte Strukturprogrammierung, also die genaue Struktur eines kombinierten Übersetzers, die Schnittstelle zur Integration der Koprozessoren mit dem auf dem Wirt laufenden Programm sowie die Partitionierung, die automatisch zwischen Software- und Hardware-Implementierung auswählt. Im nachfolgenden praktischen Kapitel 7 wird der Prototyp MODULA PIPELINE COMPILER, der zugrundeliegende Übersetzer MOCKA und die verwendete FPGA-Karte EVC1 vorgestellt, bevor Kapitel 8 Anwendungsbeispiele mit Meßergebnissen präsentiert. Schließlich werden in Kapitel 9 mögliche Erweiterungen der vorliegenden Arbeit – sowohl bezüglich der Synthese-Verfahren als auch bezüglich des zugrundeliegenden Maschinenmodells — diskutiert, und das letzte Kapitel faßt die Ergebnisse zusammen und gibt einen Ausblick auf offene Fragen.

Die Anhänge beschreiben eine Methode zur Berechnung der Gewichtsverteilung linearer Blockcodes durch rekonfigurierbare Schaltungen und ein formales Optimierungsverfahren zur Partitionierung im Hardware/Software-Codesign. Diese Verfahren wurden auch im Zusammenhang mit dieser Arbeit entwickelt.

Kapitel 2

Ein einführendes Beispiel

In diesem Kapitel betrachten wir zunächst die Bildverarbeitung als geeignetes Anwendungsgebiet, um den Nutzen rekonfigurierbarer Hardware zu verdeutlichen. Danach wird aus den Erkenntnissen dieses Beispiels die Grundstruktur eines kombinierten Übersetzers zur automatischen Erzeugung solcher Anwendungen abgeleitet und das weitere Vorgehen, wie die Verfahren dieses Übersetzers vorgestellt werden, skizziert.

2.1 Bildverarbeitung

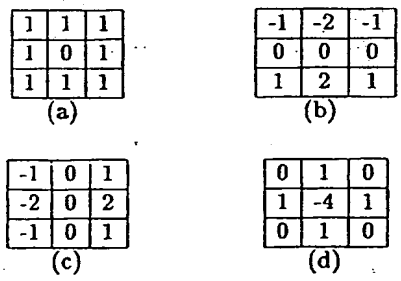


Abbildung 2.1: Masken für Bildverarbeitungs-Operatoren

Verschiedene Transformationen von Grauwertbildern werden zur Bildbearbeitung oder zur Vorverarbeitung in komplexen Bilderkennungssytemen verwendet. Dabei wird für jeden Bildpunkt des ursprünglichen Bildes ein neuer Grauwert berechnet, der nur von einer 3 × 3-Umgebung des ursprünglichen Punktes abhängt. Die Transformationen werden folgendermaßen berechnet: Jeder Punkt der 3 × 3-Umgebung wird zuerst mit einem festen Koeffizienten multipliziert, dann werden die Ergebnisse summiert und mit einer Konstanten skaliert. Gegebenenfalls muß der neue Wert

noch auf einen vorgegebenen Wertebereich (meist das Intervall von 0 bis 255) eingeschränkt werden. Die neun Koeffizienten werden oft als Matrix dargestellt und als *Maske* bezeichnet. Abbildung 2.1 stellt vier Masken dar, die verbreitete Bildverarbeitungsfunktionen durchführen. Mit dem Faktor $\frac{1}{8}$ skaliert, ersetzt die Maske (a) jeden Bildpunkt durch den Durchschnitt seiner acht umgebenden Punkte. Dieser Operator glättet das Eingangsbild und gleicht somit Bildstörungen aus. Die Sobeloperatoren der Masken (b) und (c) verstärken horizontale bzw. vertikale Kanten, führen also eine (numerische) erste Ableitung durch. Jedoch werden nur Helligkeitsänderungen in einer Richtung erkannt, da mögliche negative Werte durch die Intervalleinschränkung auf den Wert 0 abgebildet werden. Die vierte Maske (Laplaceoperator) entspricht einer zweiten Ableitung.

```
VAR P, PNEW: ARRAY[0..VERLEN-1],[0..HORLEN-1] OF CARDINAL;
...
FOR V := 1 TO VERLEN-2 DO
  FOR H := 1 TO HORLEN-2 DO
    TMP1 := P[V-1,H-1] + P[V-1,H] + P[V-1,H+1];
    TMP2 := P[V  ,H-1] +            P[V  ,H+1];
    TMP3 := P[V+1,H-1] + P[V+1,H] + P[V+1,H+1];
    PNEW[V,H] := (TMP1 + TMP2 + TMP3) DIV 8
  END
END
```

Abbildung 2.2: Bildglättung nach Abb. 2.1 (a)

Wir betrachten nun zwei konkrete MODULA-2-Programme, die aus einem Grauwertbild P der Größe VERLEN × HORLEN ein transformiertes Bild PNEW berechnen: Das erste Programm (Abbildung 2.2) glättet das Eingabebild. Eine Intervalleinschränkung ist in diesem Fall nicht nötig. Das zweite Programm (Abbildung 2.3) berechnet zunächst für jeden Punkt den horizontalen (HTMP) und vertikalen (VTMP) Kantenwert. Durch die Berechnung der Absolutwerte werden sowohl Hell/Dunkel- als auch Dunkel/Hell-Übergänge erfaßt. Schließlich kombiniert die Addition der Werte die Teilergebnisse auf einfache Weise. Dadurch können jedoch unzulässig große Werte entstehen, so daß das Ergebnis eingeschränkt werden muß. Die Abbildungen 2.4, 2.5 und 2.6 zeigen ein Grauwertbild und die Ergebnisse nach der Bearbeitung mit den Programmen.

Nun vergleichen wir die Software-Implementierung dieser Programme mit schnelleren Hardware-Realisierungen auf rekonfigurierbaren Koprozessoren. (In rekonfigurierbarer Hardware können beliebige synchrone Schaltwerke realisieren werden; ihre genauen Eigenschaften werden im nächsten Kapitel vorgestellt.)

Die Hardware-Realisierungen sollen automatisch aus den Programmen generiert werden, damit sie auch für Software-Entwickler ohne Erfahrung im Hardware-Entwurf nützlich sind. Folglich ist das erste Teilproblem, dem wir uns in dieser Arbeit wid-

```
VAR P, PNEW: ARRAY[0..VERLEN-1],[0..HORLEN-1] OF CARDINAL;
...
FOR V := 1 TO VERLEN-2 DO
  FOR H := 1 TO HORLEN-2 DO
    HTMP := (P[V+1,H-1] - P[V-1,H-1]) +
            (P[V+1,H+1] - P[V-1,H+1]) +
            2 * (P[V+1,H] - P[V-1,H]);
    VTMP := (P[V-1,H+1] - P[V-1,H-1]) +
            (P[V+1,H+1] - P[V+1,H-1]) +
            2 * (P[V,H+1] - P[V,H-1]);
    PNEW[V,H] := ABS(HTMP) + ABS(VTMP);
    IF (PNEW[V,H] > 255) THEN
      PNEW[V,H] := 255
    END
  END
END
```

Abbildung 2.3: Kantendetektion nach Abb. 2.1 (b) und (c)

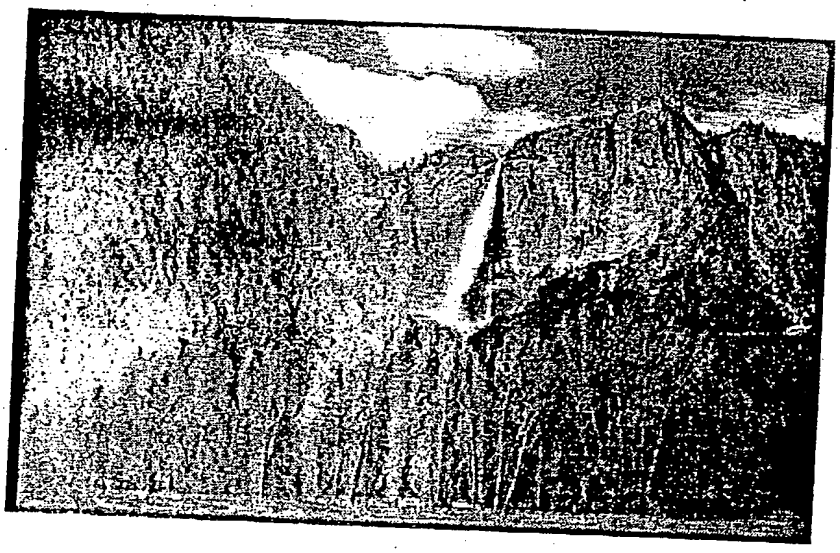


Abbildung 2.4: Originalbild

men, die Auswahl geeigneter Programmblöcke als Kandidaten für eine solche Realisierung. In unseren Beispielen sind das die jeweils innersten Schleifen, da diese in beiden Programmen für alle Bildpunkte einer Zeile dieselbe Sequenz relativ einfa-

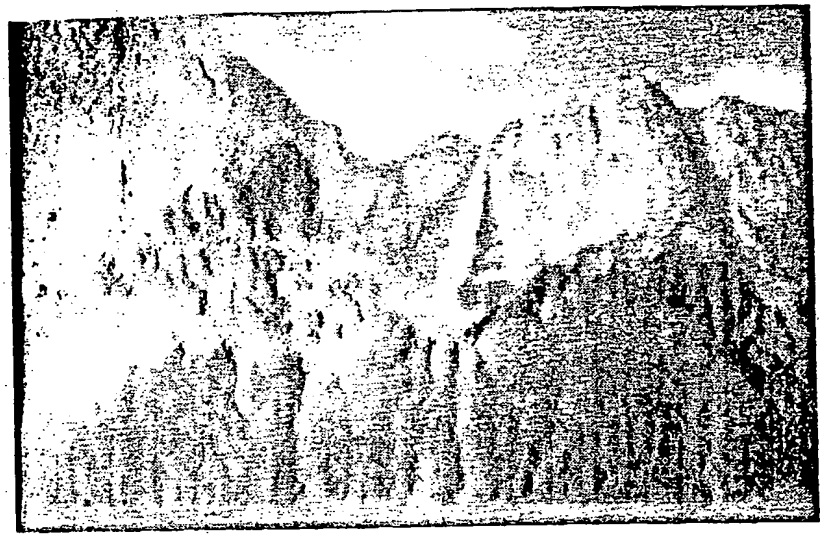


Abbildung 2.5: Geglättetes Bild

Abbildung 2.6: Kanten des Bildes

cher Berechnungen durchführen. Wir können solche inneren Schleifen *vektorisieren* und durch *Pipeline-Schaltungen* auf dem Koprozessor ausführen.

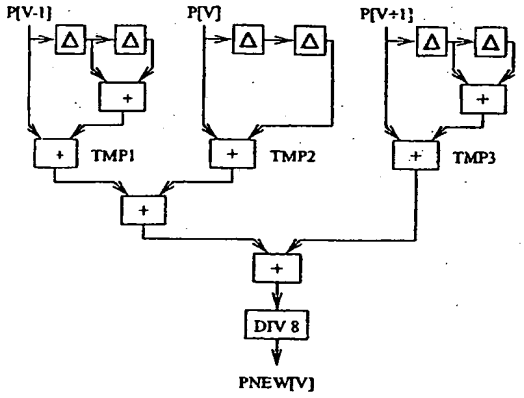


Abbildung 2.7: Pipeline-Schaltung zur Bildglättung

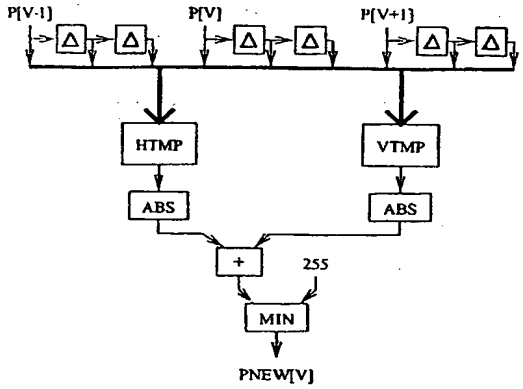


Abbildung 2.8: Pipeline-Schaltung zur Kantendetektion

Die Abbildungen 2.7 und 2.8 zeigen diese Pipelines. Die Eingaben der Schaltungen sind drei aufeinanderfolgende Zeilen (`P[V-1]`, `P[V]`, `P[V+1]`) des Eingabebildes. Die mit Δ gekennzeichneten Register speichern jeweils den letzten und vorletzten Wert der Eingaben, so daß jeweils drei nebeneinanderliegende Bildpunkte jeder Zeile vorliegen. Diese werden dann entsprechend den Vorgaben der Programme zur Ausgabe `PNEW[V]` verknüpft.[1] In Abbildung 2.8 sind die Teilschaltungen, die die Zwi-

[1] Zur Vereinfachung der Division in Hardware wurden für die Bildglättung nicht alle neun Punkte

schenergebnisse HTMP und VTMP ähnlich wie das Ergebnis PNEW[V] in Abbildung 2.7 berechnen, in einem Block zusammengefaßt dargestellt.

Offensichtlich ist das nächste — und schwierigste — Teilproblem dieser Arbeit die Koprozessor-Synthese, also die Frage, wie diese Schaltungen automatisch aus einem Programm generiert werden können. Dazu müssen auch die Analysen bestimmt werden, die für die Vektorisierung nötig sind, um aus einer sequentiellen Software-Beschreibung eine effiziente, parallele Hardware-Beschreibung zu erhalten. Danach wird geklärt, wie die Integration der Schaltungen in das auf dem Wirt laufende Programm geschehen kann.

Der Hauptvorteil einer Pipeline-Ausführung liegt darin, daß alle Operatoren parallel arbeiten können. Außerdem werden Eingabewerte und Zwischenergebnisse direkt in der Schaltung gespeichert und mit den Eingängen der nachfolgenden Operatoren verdrahtet, so daß weniger Zyklen für Speicherzugriffe und den Transport der Daten benötigt werden. Die *Strukturprogrammierung* verlagert also — im Vergleich zur gewöhnlichen Instruktionsprogrammierung — Adreßberechnungen von der Laufzeit in die Übersetzungszeit [Har96].

Dies bewirkt eine wesentlich kürzere Dauer der Pipeline-Ausführung im Vergleich zur Software-Laufzeit, wie Messungen auf einer Sun SPARCstation 10/40 mit der FPGA-Koprozessor-Karte EVC1 (siehe Abschnitt 7.1.1) gezeigt haben: Für die Bildglättung wird ein Beschleunigungsfaktor von circa 22, für die Kantendetektion sogar circa 35 erreicht.[2]

Jedoch wird der Hardware-Geschwindigkeitsvorteil durch sehr hohe Zusatzkosten erkauft: Erstens müssen die FPGAs des Koprozessors konfiguriert werden, und zweitens ist eine Übertragung der Daten vom Wirt zum Koprozessor und wieder zurück notwendig. Über die langsame Schnittstelle der EVC1-Karte, die auch keine direkte Datenübertragung zum lokalen Speicher ermöglicht, dauert die Konfigurierung des FPGA etwa 0.21 s, und die Übertragung der Bilddaten auf die Karte und zurück circa 0.20 s. Die Gesamtdauer für eine Ausführung in Hardware wird dadurch auf ein Mehrfaches der reinen Pipeline-Ausführungszeit erhöht, und die Beschleunigungsfaktoren reduzieren sich drastisch auf 1.3 bzw. 2.1. Wird in einer Messung ein Algorithmus jedoch zehnmal auf dasselbe Bild angewendet, wirken sich die Zusatzkosten nicht so stark auf die Beschleunigung aus, da insgesamt nur einmal konfiguriert und kommuniziert werden muß. Die resultierenden Faktoren sind dann acht bzw. 13.

Damit wird bereits an diesem Beispiel klar, daß die Konfigurierungs- und Kommunikationskosten einen starken Einfluß auf das Beschleunigungspotential der Koprozessoren haben und möglichst gering gehalten werden sollten. Eine Pipeline-Ausführung ist also nicht immer sinnvoll, wenn sie möglich wäre. Deshalb stellt die automatische Auswahl geeigneter Schleifen — die Hardware/Software-Partitionierung — die

---

der Umgebung verwendet, sondern nur acht. So kann die Division durch einen Schiebeoperator ersetzt werden.

[2] Die genauen Meßergebnisse werden in Abschnitt 8.2.3, Tabelle 8.3, vorgestellt, und nähere Angaben zur verwendeten Meßmethode befinden sich in Abschnitt 8.1.

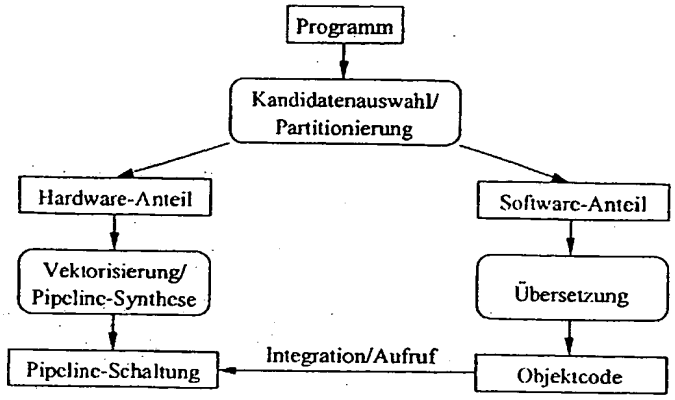


Abbildung 2.9: Grundstruktur eines kombinierten Übersetzers

letzte wichtige Fragestellung dieser Arbeit dar. Sie muß Schätzungen für die Beschleunigung durch Pipelines, aber auch für die Zusatzkosten für Konfigurierung und Kommunikation berücksichtigen.

2.2 Grundstruktur kombinierter Übersetzer

Ein kombinierter Übersetzer für einen um rekonfigurierbare Koprozessoren erweiterten Rechner benötigt folgende Funktionalität, um die anhand des im vorhergehenden Abschnitt vorgestellten Beispiels aufgezeigten Probleme zu lösen: Er wählt aus einem Eingabeprogramm in einer höheren Programmiersprache automatisch Kandidaten aus und generiert für sie Koprozessoren. Diese werden dann automatisch in das Programm integriert, und ein Hardware/Software-Partitionierer ruft sie auf, wenn dies eine Beschleunigung erwarten läßt. Daraus ergibt sich die in Abbildung 2.9 gezeigte Grundstruktur eines kombinierten Übersetzers.

Die notwendigen Übersetzer-Komponenten werden im weiteren Verlauf dieser Arbeit folgendermaßen entwickelt und vorgestellt: Da die neuen Verfahren zur Vektorisierung und Pipeline-Synthese die wichtigsten Ergebnisse der Arbeit darstellen, werden wir sie — zusammen mit der Kandidatenauswahl — in Kapitel 5 behandeln. Wir betrachten also zuerst den linken Zweig in Abbildung 2.9. Erst danach, in Kapitel 6, wenden wir uns der Hardware/Software-Integration und der Partitionierung zu, die die Koprozessor-Synthese erst zu einem kompletten Übersetzungssystem ergänzen. Die nachfolgenden Kapitel behandeln dann die praktische Implementierung und die Weiterentwicklung der Verfahren. Zunächst werden in den nächsten beiden Kapiteln jedoch die Grundlagen strukturprogrammierbarer Hardware und für uns relevante verwandte Techniken vorgestellt.

Kapitel 3

Strukturprogrammierbare Hardware

Dieses Kapitel beschreibt zunächst die wichtigsten Eigenschaften *programmierbarer Halbleiterbausteine* und ihrer Entwurfswerkzeuge sowie die Architektur *strukturprogrammierbarer Rechner*, in denen solche Bausteine eingesetzt werden. Danach werden die geeigneten Anwendungsgebiete dieser Rechner charakterisiert.

3.1 Programmierbare Logikbausteine

Dieser Abschnitt beschreibt die Architekturen programmierbarer Logikbausteine. Eine wichtige Klasse stellen die *Speicherbausteine* (RAM und ROM) dar, die hier jedoch nicht näher betrachtet werden sollen. In dieser Arbeit beschäftigen wir uns vielmehr mit Chips, die nicht zum Speichern von Daten, sondern speziell zur Realisierung anwendungsspezifischer logischer Funktionen entwickelt wurden.

3.1.1 Programmable Logic Arrays

Programmable Logic Arrays (PLAs) können eine beliebige zweistufige kombinatorische Schaltung realisieren. Die erste Stufe verknüpft beliebige Kombinationen der Eingänge (oder deren Negationen) konjunktiv, während die zweite Stufe die Ausgänge der ersten Stufe disjunktiv verknüpft. Einige Bausteine können die Ausgaben in getakteten Flipflops speichern, so daß eine Rückkopplung auf die Eingänge und damit die Realisierung endlicher Automaten möglich wird. PLAs sind in der Regel nur einmal zu programmieren.[1] Eine Einführung zu PLAs befindet sich zum Beispiel in [Wak94].

[1] Es gibt auch Bausteine, die gelöscht und ein paarmal neu programmiert werden können.

3.1.2 Field-Programmable Gate Arrays

Field-Programmable Gate Arrays (FPGAs) sind eine Weiterentwicklung der PLAs. Sie können beliebige synchrone Schaltwerke realisieren, wenn nur die Anzahl der vorhandenen FPGAs groß genug und die Taktperiode lange genug ist.[2]

FPGAs bestehen aus einem Feld *rekonfigurierbarer Logikblöcke*, die durch *schaltbare Verbindungen* beliebig kombiniert werden können. Man unterscheidet *feinkörnige FPGAs*, deren Logikblöcke zwei Eingänge verknüpfen und das Ergebnis in einem Flipflop speichern können, und *grobkörnige FPGAs*. Bei diesen bilden die Logikblöcke eine oder zwei beliebige boolesche Funktionen von mehreren (bis zu neun) Eingängen, die durch Wertetabellen realisiert werden. Deshalb werden diese FPGAs auch wertetabellen-basiert[3] genannt. Die Ausgaben der Funktionen können in bis zu zwei Flipflops gespeichert oder direkt weitergegeben werden.

Auch bei FPGAs gibt es einfach und mehrfach programmierbare Chips. Für strukturprogrammierbare Rechner sind nur FPGAs geeignet, die beliebig oft und in der Schaltung ohne spezielle Geräte programmiert werden können. In diesem Fall sprechen wir von *rekonfigurierbarer* oder *strukturprogrammierbarer Hardware*. Denn im Gegensatz zur Instruktionsprogrammierung eines Prozessors (in der Zeit) können in ein FPGA Strukturen (in der Fläche) programmiert werden, die die auszuführende Berechnung definieren. Die FPGA-Konfiguration ist also nicht mehr eine in der Halbleiter-Fabrik oder beim Gerätehersteller statisch festgelegte Schaltung (Hardware), sondern entspricht eher einer dynamischen Programmierung (Software), da sie beliebig oft und direkt im Rechner des Anwenders durchgeführt werden kann.

Bevor ein rekonfigurierbares FPGA arbeitet, muß sein aus statischem RAM bestehender Konfigurationsspeicher geladen werden. Diese *Konfigurierung* dauert bei den verbreiteten FPGAs circa 10 - 100 Millisekunden und bestimmt die danach vorhandene Anwenderlogik. Wird eine andere Schaltung benötigt, kann das FPGA wieder umkonfiguriert werden. Da der Konfigurationsspeicher flüchtig ist, muß das FPGA auch nach dem Abschalten der Stromversorgung neu konfiguriert werden. Einige FPGA-Familien können auch partiell rekonfiguriert werden, während andere Schaltungsteile weiterarbeiten. Bei FPGAs der neuen Serie XC6200 [KW95] der Firma Xilinx kann zusätzlich direkt über eine schnelle Schnittstelle auf die Werte der internen Flipflops zugegriffen werden, weshalb diese Bauteile besonders für Koprozessor-Anwendungen in strukturprogrammierbaren Rechnern geeignet sind. Diese FPGAs können auch sehr schnell rekonfiguriert werden: Beim XC6216 dauert eine komplette Konfigurierung nur circa 100 Mikrosekunden.

Abbildung 3.1 zeigt den Aufbau eines grobkörnigen FPGA der Serie XC4000 der Firma Xilinx [Xil94a]. Zwischen den CLB (configurable logic block) genannten Lo-

[2] Die *Taktperiode* einer synchronen Schaltung muß mindestens so lange wie die längste kombinatorische *Signalverzögerung* sein. Der Kehrwert der Periode ist die *Taktfrequenz* und entspricht dem *Durchsatz* der Schaltung (also der Rate, mit der die Ergebnisse ausgegeben werden), falls in jedem Takt ein Ergebnis erzeugt wird.

[3] engl. lookup-table- oder LUT-based

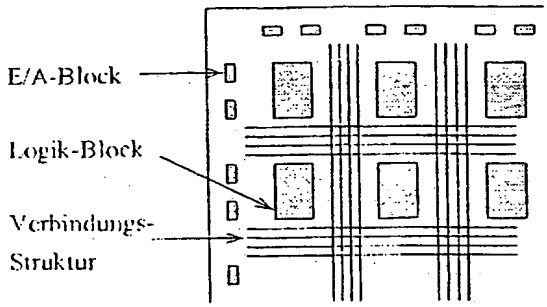


Abbildung 3.1: Aufbau eines grobkörnigen Xilinx FPGA

gikblöcken befinden sich Verdrahtungskanäle, die sich in einzelne Segmente aufteilen und über Kreuzverteiler kombinieren lassen. Die Pins des Chips sind über Ein-Ausgabe-Blöcke (IOB, in/out block) mit dem Inneren des FPGA verbunden. Die IOBs verstärken die Pin-Signale und enthalten je ein Eingabe- und ein Ausgabe-Flipflop, um beispielsweise asynchrone Eingänge zu synchronisieren.

Eine wesentliche technologische Voraussetzung für diesen Bausteintyp war die Realisierung von schaltbaren Verbindungen in CMOS-Technologie, mit der sich erst die Verdrahtung eines beliebigen Schaltwerks realisieren läßt.

Die ersten FPGAs hatten nur eine Komplexität von wenigen tausend Gatteräquivalenten. Sie waren daher nur als flexible Implementierung für unstrukturierte Schaltungsteile ("glue logic") einzusetzen. Die Logik in heutigen FPGAs entspricht bis zu 100.000 Gattern. Damit lassen sich auch ganze Funktionseinheiten in FPGAs realisieren. Die elementaren Verarbeitungseinheiten dieser FPGAs sind weiterhin einzelne Bits, die für Daten mit größerer Wortlänge zusammengefaßt werden müssen. Jedoch unterstützen einige FPGA-Familien auch arithmetische Operatoren in besonderer Weise, wie etwa die "dedicated carry logic" für Zähler und Addierer der Xilinx XC4000-Serie. Jedoch benötigen kompliziertere Operatoren wie Multiplizierer und Dividierer sowie Gleitkommaoperatoren so viele Gatter, daß eine Realisierung auf heutigen FPGAs meist nicht in Frage kommt. Dies ist insbesondere der Fall, wenn die rekonfigurierbare Logik zur Beschleunigung von Standardprozessoren dienen soll, die über hochoptimierte Arithmetik-Einheiten verfügen.

Neuere FPGA-Architekturen wie rDPA/Xputer [HKR94] oder RaPiD [ECF96] wurden nicht zur Realisierung beliebiger Logik entworfen, sondern speziell für arithmetik-intensive Anwendungen, etwa in der digitalen Signalverarbeitung. Sie verfügen über breitere Datenpfade, verarbeiten also ganze Wörter als elementare Einheit. Diese *wort-orientierten FPGAs* kombinieren die Vorteile der Strukturprogrammierung mit leistungsfähigen, nicht konfigurierbaren Operatoren und ALUs. Sie eignen sich als alternative Zielplattform für die in dieser Arbeit vorgestellte Programmiermethodik.

Zur Verknüpfung mehrerer FPGAs werden auch *Field-Programmable Interconnect Devices (FPIDs)* verwendet. Sie basieren auch auf schaltbaren Verbindungen und ermöglichen es dem Entwickler, die Pins der angeschlossen FPGAs beliebig zu verdrahten. Dies entlastet die meist knappen internen Verdrahtungsressourcen der FPGAs.

3.2 Entwurfswerkzeuge für FPGAs

Der Schaltungsentwurfs-Prozeß für FPGAs unterscheidet sich nicht grundsätzlich von dem für ASICs (application specific integrated circuits). Er wird durch Entwurfswerkzeuge[4] unterstützt. Diese spezifizieren die Funktion auf höheren Ebenen als die frühen Silicon Compiler.

3.2.1 High-Level-Synthese

Unter High-Level-Synthese [MR92, Cam90, GDWL92, CW91] versteht man den automatischen Entwurf von meist synchronen Register-Transfer-Strukturen (RT-Strukturen) aus *algorithmischen Verhaltensbeschreibungen*. Diese Spezifikationen sind Programmen in höheren Software-Programmiersprachen sehr ähnlich. Deshalb enthalten High-Level-Synthese-Systeme auch Komponenten gewöhnlicher Übersetzer, und aus dem Software-Bereich bekannte Optimierungsverfahren können verwendet werden. Im Gegensatz zu normalen Programmiersprachen können diese *Hardware-Beschreibungssprachen*, z. B. die genormte Sprache VHDL[5] [IEE, LSU89], auch nicht-funktionale Aspekte wie das Zeitverhalten einer Schaltung ausdrücken. In den meisten Fällen können auch parallele Prozesse spezifiziert werden, und in einigen Sprachen dürfen Verhaltensbeschreibungen und strukturelle Beschreibungen kombiniert werden. So können unterschiedliche Schaltungsteile auf verschiedenen Abstraktionsniveaus spezifiziert werden.

Die Schwierigkeit der High-Level-Synthese liegt weniger darin, *irgendeine* verhaltensgleiche RT-Struktur für die algorithmische Eingabe zu erzeugen, sondern darin, eine *effiziente* Struktur zu finden, die die Geschwindigkeit und den Flächenbedarf optimiert. Jedoch ist die optimale Umsetzung einer Verhaltens- in eine Strukturbeschreibung bei beschränkten Ressourcen ein NP-hartes Problem. Deshalb wird es in mehrere Teilprobleme zerlegt, für die sich einfacher Lösungen finden lassen. Die Optimalität des Gesamtergebnisses ist damit jedoch nicht mehr gewährleistet. Gewöhnlich wird die Synthese in folgenden Einzelschritten durchgeführt:

- Zunächst wird die Eingabesprache in ein *internes Zwischenformat* übersetzt. Es besteht aus Graphen, die den Steuerfluß des Programms (wie im Kontrollfluß-Graphen eines optimierenden Übersetzers) und den Datenfluß

[4] engl. computer aided design (CAD) oder electronic design automation (EDA) tools
[5] VHSIC Hardware Description Language

repräsentieren. Meist wird für jeden Basisblock ein separater Datenfluß-Graph erzeugt, der die lokalen Datenabhängigkeiten widerspiegelt. Bei der Erstellung des Zwischenformats werden *optimierende Programmtransformationen* durchgeführt. Dies sind gewöhnliche Übersetzer-Optimierungen wie Konstantenpropagation, Elimination toten Codes und gemeinsamer Teilausdrücke oder Schleifenausrollen.

Zusätzlich können auch hardware-spezifische Transformationen durchgeführt werden. Beispielsweise verschmilzt eine Kontrollfluß-Datenfluß-Transformation [GDWL92, Abschnitt 5.7.3] mehrere Basisblöcke, die eine bedingte Zuweisung repräsentieren. Dabei werden die Datenfluß-Graphen kombiniert, indem Multiplexer hinzugefügt werden, die den richtigen Wert der Variablen auswählen. Dieser größere Datenfluß-Graph ermöglicht später mehr boolesche Optimierungen.

- Die *Ablaufplanung* (engl. scheduling) ordnet jeder Operation einen Zeitschritt (d. h. einen Takt) zu. Sie muß vorhandene Datenabhängigkeiten und den vorgegebenen Kontrollfluß berücksichtigen, so daß der mögliche Parallelitätsgrad meist recht beschränkt ist. Schleifen müssen, wenn sie nicht ausgerollt werden konnten, sequentiell abgearbeitet werden. Deshalb wurden erweiterte Verfahren zur Synthese von Pipelines für Schleifen entwickelt. Diese werden in Abschnitt 4.3.3 vorgestellt.

  Da die Ablaufplanung eigentlich nicht ohne Berücksichtigung der zur Verfügung stehenden Hardware-Ressourcen durchgeführt werden kann, werden verschiedene Heuristiken zum Schätzen der benötigten Ressourcen verwendet.

- Die *Bereitstellung von Ressourcen* (engl. allocation) legt die verwendeten Verarbeitungseinheiten (arithmetische und logische Operatoren), Speicherzellen (Register und RAM) und Verbindungen (Busse) fest. In vielen Systemen werden diese auch vom Benutzer vorgegeben. Die Bandbreite der möglichen Lösungen reicht von einer universellen Verarbeitungseinheit für alle Operationen (minimaler Platzbedarf, aber sequentielle Ausführung) bis zu einem eigenen Operator für jede Operation (maximale Parallelität, aber sehr hoher Platzbedarf). Wegen der fehlenden Mehrfachnutzung der Ressourcen ist diese "direkte Übersetzung" [MR92] jedoch im allgemeinen nicht sinnvoll.

- Im nächsten Schritt erfolgt die *Zuordnung* (engl. assignment) von Operationen zu Operationseinheiten, von Variablen zu Speicherzellen und von Kommunikationen zu Bussen. Die Zuordnung wird oft mit der Bereitstellung der Ressourcen kombiniert und definiert die Struktur der Schaltung, den *Datenpfad*.

- Schließlich wird ein *Steuerwerk* (engl. controller) synthetisiert, das die Ausführung der Operationen durch die Komponenten des Datenpfads gemäß dem Ablaufplan steuert. Dieses ist ein Moore-Automat, der in jedem Zustand (Zeitschritt) ein Steuerwort an den Datenpfad leitet und aus dem aktuellen Zustand und den Ausgaben des Datenpfads den Folgezustand berechnet.

- Die Komponenten des Datenpfads und des Steuerwerks (auf Register-Transfer-Ebene) werden als *Netzliste*, d. h. in einem textuellen Standard-Austausch-Format, das die Graphen repräsentiert, ausgegeben.

Wir nennen diesen Ansatz im folgenden *Standard-High-Level-Synthese*. Seine Allgemeinheit und die vielen Freiheitsgrade erschweren die optimale Synthese für viele Anwendungen. Allein die Auswahl der anzuwendenden Programmtransformationen oder die Einschränkungen bei der Ressourcen-Bereitstellung haben einen großen Einfluß auf das Syntheseergebnis. Deshalb wurden bereichsspezifische Synthesesysteme (etwa speziell für Prozessorarchitekturen) entwickelt, die bessere Ergebnisse als die allgemein einsetzbaren Systeme liefern.

Wegen der durch FPGAs gegebenen Möglichkeit, ohne hohe Kosten mit digitalen Schaltungen zu experimentieren, wurden auch Werkzeuge entwickelt, die verbreitete Software-Programmiersprachen direkt zur automatischen Synthese synchroner Schaltungen verwenden. Dies soll die Akzeptanz für Software-Entwickler erhöhen. Jedoch kann für eine effiziente Hardware-Synthese nicht ganz auf die zusätzlichen Komponenten der Hardware-Beschreibungssprachen verzichtet werden. So ist der Flächenbedarf und die Geschwindigkeit der erzeugten Schaltungen meist nicht zufriedenstellend, da nur einfache High-Level-Synthese-Verfahren verwendet werden. Z. B. findet im Transmogrifier C Compiler [Gal95] keine Mehrfachnutzung der Ressourcen und keine automatische Ablaufplanung statt. Auch verändert der auf einer stark eingeschränkten Teilmenge von C basierende Übersetzer die Semantik der Sprache und benötigt zur parallelen Ausführung mehrerer Threads zusätzliche Sprachkonstrukte.

3.2.2 Logiksynthese

Die Logiksynthese verarbeitet Netzlisten weiter. Diese können entweder die Ausgabe einer High-Level-Synthese sein oder direkt vom Entwickler eingegeben werden. Dazu werden *graphische Eingabewerkzeuge*[6], mit denen Schaltungskomponenten ausgewählt und verknüpft werden können, oder strukturelle Beschreibungen in einer Hardware-Beschreibungssprache verwendet.

Die Netzliste wird zuerst *technologieunabhängig* optimiert. D. h., es werden allgemeingültige logische Transformationen durchgeführt, die den Hardware-Bedarf oder die Signalverzögerung reduzieren. Die anschließende *technologieabhängige* Weiterverarbeitung geschieht spezifisch für eine bestimmte FPGA-Familie. Sie unterscheidet sich stark vom ASIC-Entwurf, da die Komponenten der Netzliste nicht auf Standardzellen oder einzelne Gatter, sondern auf die Logikblöcke der jeweiligen FPGA-Architektur abgebildet werden. Dabei werden einzelne Gatter und Flipflops auf Logikblöcke verteilt und komplexere Operatoren durch bereits optimierte Module aus einer Bibliothek für die entsprechende FPGA-Familie ersetzt.

Im nächsten Schritt werden die Logikblöcke plaziert und verdrahtet. Dabei können wegen der begrenzten Verdrahtungskanäle manchmal auch vollständig plazierte

[6] engl. schematic entry tools

Schaltungen nicht oder nur mit sehr hohen Signalverzögerungen verdrahtet werden. Wegen der exponentiellen Komplexität der Berechnungen kann die optimale Plazierung und Verdrahtung nur mit heuristischen Methoden angenähert werden. Dies führt zu sehr langen Rechenzeiten (bis zu mehreren Stunden für ein FPGA auf einer schnellen Workstation) und unvorhersagbaren Resultaten. Um eine Verdrahtung mit niedrigen Verzögerungen zu erhalten, muß oft auf eine hohe Auslastung der verfügbaren Logikblöcke verzichtet werden. Wegen dieser Probleme werden Schaltungen auch teilweise manuell plaziert, um eine höhere Bausteinauslastung und kürzere Taktperioden zu erreichen.

Im Gegensatz zum ASIC-Entwurf ist für FPGAs ein viel geringerer Aufwand zur *Simulation* der Schaltungen nötig. Denn die Schaltungen werden nicht in hoher Stückzahl — mit entsprechenden Kosten bei einem Fehler — gefertigt, sondern zunächst nur mit einem FPGA getestet. Im Falle eines Fehlers fallen bei einmalprogrammierbaren FPGAs nur geringe und bei rekonfigurierbaren FPGAs gar keine Materialkosten an. Die Hardwaresimulation kann also weitgehend durch Testen im FPGA ersetzt werden.

Für strukturprogrammierbare Rechner wurden einige auf FPGAs zugeschnittene Hardware-Beschreibungssprachen entwickelt, die vor allem die gezielte Plazierung mehrfach vorkommender Schaltungskomponenten erleichtern. Beispiele dafür sind *LDG (Logic Description Generator)* [GKLM90] und *PERLE1DC* [BT94]. Wie oben bereits erwähnt, ist dies jedoch nur nötig, wenn man eine sehr hohe Auslastung der FPGAs erreichen will.

3.3 Strukturprogrammierbare Rechner

Für Rechnerarchitekturen, die programmspezifische Hardware-Unterstützung durch konfigurierbare, FPGA-basierte Koprozessoren realisieren, hat sich bisher keine einheitliche Bezeichnung durchgesetzt. Im Englischen wird diese Rechnerarchitektur meistens nach einer gleichnamigen Workshop-Serie [BP93, BP94, AP95, PA96] *(FPGA-based) Custom Computing Machine (CCM)* — siehe auch [Har95] — oder *Programmable Active Memory (PAM)* [BRV89] genannt. Außerdem wird noch der Begriff *rekonfigurierbare Architektur* verwendet, der jedoch zur Verwechslung mit auf höherer Ebene rekonfigurierbaren Rechnern — etwa Parallelrechnern mit variabler Verbindungstopologie — führen kann. Wir verwenden deshalb in dieser Arbeit den Begriff *strukturprogrammierbare Rechner* (SP-Rechner), da er die wesentliche Eigenschaft dieser Systeme — die Strukturprogrammierung — enthält. FPGAs stellen zwar die heutige Implementierungstechnologie dar, sind aber nicht prinzipiell zur Strukturprogrammierung nötig und sollten daher kein Namensbestandteil der zugehörigen Rechnerarchitektur sein.

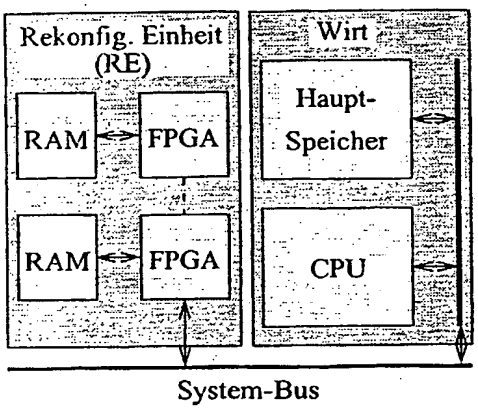


Abbildung 3.2: Architektur eines strukturprogrammierbaren Rechners

3.3.1 Wirt und Rekonfigurierbare Einheit

SP-Rechner bestehen aus einem konventionellen, mikroprozessor-basierten *Wirt* und einer *rekonfigurierbaren Einheit* (RE). Diese enthält neben einem oder mehreren FPGAs meist *lokalen Speicher*. Abbildung 3.2 zeigt diese Basisarchitektur. Bisher realisierte SP-Rechner unterscheiden sich stark in der Anzahl, Größe und Anordnung der FPGAs und des lokalen Speichers sowie in der Realisierung der Schnittstelle zum Wirt [Guc]. Zum Beispiel ist bei manchen eine direkte Datenübertragung vom Hauptspeicher in den lokalen Speicher der RE möglich, was bei anderen nur über CPU und FPGAs möglich ist. SP-Rechner wurden anhand dieser Merkmale von Guccione [GG95a] und Hartenstein [HBK96] klassifiziert: Neben kleinen Koprozessor-Karten ohne lokalen Speicher (wie dem PRISM-Hardware-Prototyp [AS93]) gibt es große rekonfigurierbare Supercomputer, die den Wirt nur als Vorrechner benötigen und für einige Anwendungen die Leistungsfähigkeit aller anderen Parallel- und Vektorrechner übertreffen. Beispiele hierfür sind Splash/Splash 2 [GHK+91, BAK96] und DECPeRLe [BRV93a, VBR+96]. Einige Architekturen setzen neben FPGAs auch FPIDs zur flexibleren Verbindung der FPGAs ein. Dies ist beispielsweise beim Virtual Computer [Cas93] der Fall. Dessen stark verkleinerte Version, der Engineer's Virtual Computer EVC1 [TCS94, CTS95], wird kommerziell von der Virtual Computer Corporation vertrieben. Diese SBUS-Karte mit einem Xilinx XC4013-FPGA und 2 MB lokalem SRAM wurde für die praktischen Experimente in dieser Arbeit verwendet.

Eine engere Kopplung von Mikroprozessor und FPGA wird in einigen Forschungsprojekten untersucht. Die Kopplung geschieht entweder über den Koprozessor-Port des Prozessors (mit dem FPGA auf derselben Platine) oder sogar durch die Integration der beiden Elemente auf demselben Chip [DeH94].

Bei einigen Architekturen kann die RE auch direkt mit Peripheriegeräten kommuni-

zieren. Z. B. können Splash 2 und DECPeRLe direkt Videodaten von einer Kamera verarbeiten oder an einen Monitor ausgeben. Diese Möglichkeiten, die aus dem SP-Rechner ein eingebettetes System (zur Echtzeitverarbeitung) machen, werden in dieser Arbeit nicht näher betrachtet. Jedoch sind die vorgestellten Verfahren prinzipiell auch zur direkten Bearbeitung von Daten aus Peripheriegeräten geeignet.

3.4 Anwendungen

In diesem Abschnitt charakterisieren wir zunächst die Eigenschaften, die darüber entscheiden, ob Berechnungen eher für die Ausführung in einem Mikroprozessor (in Software) oder in den FPGAs der RE (in Hardware) geeignet sind. Daraus ergeben sich die Anwendungsgebiete, in denen SP-Rechner bereits erfolgreich eingesetzt wurden. Im nächsten Kapitel werden dann Methoden diskutiert, mit denen sich solche Anwendungen automatisch aus abstrakten Spezifikationen generieren lassen.

SP-Rechner-Anwendungen wurden außer auf den bereits erwähnten Workshops über *FPGA-based Custom Computing Machines* — vor allem auf den FPL-Workshops *(Field-Programmable Logic and Applications)* [ML91, GH92, ML93, HS94, ML95, HG96] vorgestellt.

3.4.1 Mikroprozessoren

Betrachten wir zuerst die Eigenschaften heutiger Mikroprozessoren: Sie bearbeiten Daten immer *wortweise* gemäß den Befehlen eines Maschinenprogramms. Da der ausgeführte Befehl in jedem Takt (im wesentlichen ohne zusätzliche Kosten) wechseln kann, sind Mikroprozessoren auch für sehr *irreguläre* Berechnungen (mit häufig wechselnden Befehlen) geeignet. Dabei ist der Befehlssatz des Prozessors für *typische* Anwendungen optimiert: Arithmetische und logische Operationen, die in gewöhnlichen Programmen häufig auftreten, können von dem Prozessor direkt ausgeführt werden. Für selten vorkommende Operationen lohnt es sich jedoch nicht, spezielle funktionale Einheiten auf dem Prozessor zu implementieren. Sie müssen daher umständlich mit den vorhandenen Maschinenbefehlen realisiert werden.

3.4.2 Rekonfigurierbare Hardware

Für den effizienten Einsatz rekonfigurierbarer Hardware sind zwei Charakteristika entscheidend, die wir im folgenden betrachten.

Spezielle kombinatorische Funktionen

Im Gegensatz zu Mikroprozessoren können (gewöhnliche, nicht wort-orientierte) FPGAs beliebige logische Funktionen direkt in Hardware realisieren. Solche Funktionen, die nicht im Befehlssatz eines Standard-Prozessors auftauchen und durch

ein rein kombinatorisches Schaltnetz realisiert werden können, wollen wir im folgenden *spezielle kombinatorische Funktionen* nennen. Beispiele dafür sind die folgenden Funktionen auf Bitvektoren: Gewichtsberechnung, Hamming-Distanz, Bit-Permutierung, Finden der ersten Eins, sowie fehlerkorrigierende Kodierung und Dekodierung. Sie werden durch spezifische Schaltungen im Vergleich zur Software-Ausführung wesentlich beschleunigt. Da sie aber eben nur selten in Programmen vorkommen, lohnen sich die Kosten für die Konfigurierung der RE nur in wenigen Fällen. Eine wesentliche Beschleunigung eines ganzen Programms kann also allein durch spezielle kombinatorische Funktionen kaum erreicht werden.

Feinkörnige Parallelität

Wegen der obengenannten Einschränkungen wird in den meisten Anwendungen eine weitere Eigenschaft der FPGAs ausgenützt: Alle Hardware-Operatoren können parallel arbeiten. Dies ist jedoch nur dann hilfreich, wenn die Struktur der Berechnung eine *massive, feinkörnige Parallelität* ermöglicht und sehr *regulär* ist, d. h. die gleiche Folge von Befehlen auf große Datenmengen anwendet. Denn für häufig wechselnde Befehle müßten die entsprechenden Operatoren entweder gleichzeitig in der Hardware allokiert werden, was die effektive Nutzung der FPGAs stark reduzieren würde, oder ein Teil der RE müßte häufig umkonfiguriert werden, was jedoch jeweils hunderttausende Zyklen dauert.

Daher sind solche Anwendungen besonders geeignet, die sich durch Pipelines ausführen lassen. Denn wie beim VLSI-Entwurf ermöglichen Pipelines leistungsfähige Schaltungen mit einem hohen Durchsatz. Bei der Realisierung in rekonfigurierbarer Logik sollten jedoch aus den in Abschnitt 3.1.2 genannten Gründen möglichst nur Berechnungen mit einfachen arithmetischen, logischen oder speziellen kombinatorischen Operationen mit kleinen Wortlängen vorkommen. Dies ist bei einigen Algorithmen der digitalen Signal- und Bildverarbeitung (z. B. Filter und Faltungen auf Festkommadaten, diskrete Cosinus- und Hough-Transformation [VBR+96, BRV93b]) oder bei neuronalen Netzen der Fall. Für diese Aufgaben eignen sich auch die neuen wort-orientierten FPGAs. Sie — oder eine größere Anzahl konventioneller FPGAs — ermöglichen es auch, arithmetik-intensivere Aufgaben wie die Berechnung einer FFT mit Gleitkommazahlen auf SP-Rechnern durchzuführen. Da Pipelines zum Entwurf effizienter Schaltungen sehr wichtig sind, gehen wir im nächsten Kapitel näher auf Verfahren zur automatischen Pipeline-Synthese ein.

Einen hohen Parallelitätsgrad ermöglichen auch systolische Felder, die eine Verallgemeinerung des Pipeline-Prinzips darstellen. In diesen Feldern fließen Datenströme in mehreren Dimensionen und in entgegengesetzten Richtungen zwischen kleinen, gleichartigen Prozessorelementen. Zum Beispiel verwendet eine der bekanntesten und erfolgreichsten SP-Rechner-Anwendungen — der Vergleich von menschlichen Genomen [BAK96] — ein eindimensionales systolisches Feld. Es vergleicht eine neue Gen-Sequenz hochparallel mit einer großen Datenbank bekannter Sequenzen und berechnet ihre Ähnlichkeit als "Editierdistanz". Diese Distanz entspricht der mini-

malen Zahl der Mutationen, die nötig wären, um die beiden Sequenzen ineinander zu überführen. Da das zugrundeliegende "Alphabet" der Chromosomen nur aus vier Elementen besteht, lassen sich auf einer RE sehr viele kleine Prozessorelemente realisieren. Wegen der guten Parallelisierbarkeit und der kleinen Wortlängen können Systeme zur Mustererkennung und zum Vergleich von Zeichenketten (etwa zur Datenkompression) prinzipiell gut auf SP-Rechnern implementiert werden.

Desweiteren wurden SP-Rechner erfolgreich für die Langzahlarithmetik eingesetzt. Dies ist kein Widerspruch zu der obengenannten Forderung nach kurzen Wortlängen. Denn es wurden bit-serielle Addierer (mit einer effektiven Wortlänge von einem Bit) und parallel-serielle Multiplizierer implementiert. Ein Nachteil dieser Lösung ist jedoch, daß bei der Übertragung zwischen Prozessor und RE immer zwischen bit-paralleler und bit-serieller Datenrepräsentation transformiert werden muß.

3.4.3 Kooperation im strukturprogrammierbaren Rechner

Die meisten echten Anwendungen sind weder völlig irregulär noch völlig regulär. Deshalb werden die geeigneten regulären Anteile auf der RE ausgeführt, während der Rest auf dem Wirt läuft. Dieser steuert auch die Berechnungen auf der RE. Mit Hilfe seines Betriebssystems werden die FPGA-Konfigurationen auf dem Hintergrundspeicher des Wirts gespeichert, und beim Aufruf einer Anwendung wird die RE vom Wirt konfiguriert. Danach läuft ein Programm auf dem Wirt ab, das die Funktionen auf der RE aufruft. Falls die Anwendung dies zuläßt, können Wirt und RE auch parallel arbeiten.

Der Wirt wird auch zur Speicherung der Anwendungsdaten gebraucht. Dabei erweist sich die Kommunikations-Bandbreite zwischen Wirt und RE oft als Flaschenhals. Deshalb muß das Berechnungs/Kommunikations-Verhältnis in der RE möglichst groß sein. Da von der RE der direkte Zugriff auf den Hauptspeicher des Wirt nur langsam oder gar nicht möglich ist, wird der lokale Speicher der RE als Cache verwendet und dient in den meisten Fällen zur Speicherung von Zwischenergebnissen oder Wertetabellen. Falls jedes FPGA einen eigenen lokalen Speicher besitzt, ist so auch der verteilte, gleichzeitige Zugriff auf mehrere Speicherbänke möglich.

Der Kommunikations-Flaschenhals tritt weniger in Erscheinung, wenn externe Datenströme direkt mit hoher Bandbreite an die RE gekoppelt sind. Der Wirt muß dann jedoch zusätzlich die Peripheriegeräte steuern. Eine Sonderrolle nehmen auch SP-Rechner ein, die zu Überwachungs- und Steuerungs-Aufgaben eingesetzt werden. Bei ihnen ist die RE direkt mit den Sensoren und Aktoren gekoppelt. Die Anforderungen an die Steuerung sind meist einfach, müssen aber in Echtzeit erfüllt werden. Dies ist direkt in Hardware effizient zu realisieren. Zusätzlich können die FPGAs Eingabedaten direkt vorverarbeiten, bevor sie an den Wirt übertragen werden.

Kapitel 4

Verwandte Techniken

Dieses Kapitel stellt bekannte Techniken vor, die für die in dieser Arbeit entwickelten Übersetzungsmethoden für SP-Rechner relevant sind.

Zunächst diskutiert der nächste Abschnitt Ansätze zur kombinierten Spezifikation des auf dem Wirt ablaufenden Programms (des *Software-Anteils* der Anwendung) und der in der RE ausgeführten Funktionen (des *Hardware-Anteils*). Danach erörtert Abschnitt 4.2 Techniken zur automatischen Partitionierung einer Anwendung in Hardware- und Software-Anteile. Abschnitt 4.3 ist Pipeline-Synthese- und -Optimierungsverfahren gewidmet, und Abschnitt 4.4 stellt Vektorisierungsverfahren für sequentielle Programme vor, die bisher nur für Vektor- und Parallelrechner eingesetzt wurden, aber in dieser Arbeit für Hardware-Pipelines erweitert werden.

4.1 Ansätze zur kombinierten Programmierung

Die meisten Anwendungen für SP-Rechner werden bis heute manuell entworfen. Dabei werden die Schaltungen für die RE (Hardware-Anteil) mit konventionellen Entwurfswerkzeugen (siehe Abschnitt 3.2) unabhängig vom auf dem Wirt laufenden Programm entworfen. Danach erfolgt die manuelle Integration der Schnittstelle zwischen Software und Hardware. Da dieses Vorgehen vom Entwickler sowohl Kenntnisse im Hardware-Entwurf als auch in der Software-Entwicklung erfordert und außerdem zeitaufwendig und fehleranfällig ist, wird es hier nicht weiter betrachtet.

Vielmehr stellen wir Programmiermethoden vor, die eine ganze Anwendung, also ihren Software- und ihren Hardware-Anteil, kombiniert in einer höheren Programmiersprache spezifizieren. Sie ermöglichen auch Software-Entwicklern die Verwendung strukturprogrammierbarer Rechner. Jedoch sind die bekannten Verfahren nur für wenige Programme oder für sehr eingeschränkte Problemklassen sinnvoll einzusetzen, so daß sie bisher nur in Forschungs-Prototypen implementiert wurden.

4.1.1 Sequentielle Sprachen

Die Verwendung einer verbreiteten sequentiellen Programmiersprache ist für den Entwickler am einfachsten, da das Programm zunächst komplett in Software auf einem Mikroprozessor getestet werden kann, bevor es durch rekonfigurierbare Hardware beschleunigt wird.

Dies ist etwa bei *PRISM*[1] [AS93, Ath92] und *PRISM-II* [AWG94] der Fall. Beides sind vollständig implementierte Programmiersysteme für SP-Rechner, die Standard-C als Eingabesprache verwenden und für vom Programmierer definierte Funktionen Koprozessoren generieren, die auf eine RE ausgelagert werden. Während eines Koprozessor-Aufrufs wartet der Hauptprozeß (Software). Das ältere PRISM-System führt für die C-Funktionen eine einfache High-Level-Synthese durch, die alle Operatoren in einem Kontrollschritt anordnet. So werden nur kombinatorische Schaltnetze synthetisiert, die kein Steuerwerk benötigen. Es können aber auch nur kleine Funktionen mit statisch (zur Übersetzungszeit) ausrollbaren Schleifen behandelt werden. Dies sind im wesentlichen die in Abschnitt 3.4.2 definierten speziellen kombinatorischen Funktionen. Durch die Verbindung mit einer nachfolgenden logischen Minimierung für Xilinx FPGAs werden effiziente Schaltnetze für die Funktionen realisiert. Die Koprozessoren führen also in gewissem Sinne eine *Befehlssatz-Erweiterung* des Wirts durch.

Dagegen sind die von PRISM-II erzeugten Koprozessoren synchrone Schaltungen mit Steuerwerk. So können auch beliebige Schleifen behandelt werden. Der Parallelität sind jedoch durch die Datenabhängigkeiten der sequentiellen Eingabesprache Grenzen gesetzt. Es wird ein zwar spezialisierter, aber im wesentlichen immer noch sequentieller Koprozessor generiert. Deshalb ist in jedem Takt meist nur ein Operator aktiv.

Folglich weisen die Hardware-Koprozessoren bei PRISM und PRISM-II nur in wenigen Fällen genügend Parallelität — wie in Abschnitt 3.4.2 gefordert — auf, um trotz der hohen Kosten für die Konfigurierung und die Kommunikation mit dem Wirt eine Beschleunigung zu erhalten. Der Einsatzbereich dieser Systeme ist daher stark eingeschränkt.

4.1.2 Parallele Sprachen

Enthält eine Eingabesprache explizit parallele Konstrukte, kann auch die Parallelität der Hardware besser genutzt werden. Dies ist bei den in diesem Abschnitt vorgestellten Verfahren der Fall. Jedoch kann ein paralleles Programm nicht so einfach wie ein sequentielles in Software getestet werden.

Außerdem legen die meisten parallelen Programmiersprachen ein bestimmtes Ausführungsmodell fest. Da FPGAs generische Hardware-Ressourcen zur Verfügung stellen, können sie — je nach Konfigurierung — diese unterschiedlichen parallelen

[1] Processor Reconfiguration through Instruction Set Metamorphosis

Ausführungsmodelle auch unterstützen. Dies ist für eine effiziente Implementierung spezieller Anwendungsklassen sinnvoll, schränkt aber die Vielfalt der möglichen Architekturen und den Anwendungsbereich der Verfahren ein.

Beispielsweise spezifizieren die auf OCCAM [PL91] und der verwandten Sprache PROMELA [WOB93] aufbauenden Systeme die Anwendung als *kommunizierende Prozesse*. Da diese schon als relativ unabhängige Einheiten spezifiziert werden, bieten sie sich als natürliche Einheit zur Aufteilung in Hardware und Software an. Denn jeder Prozeß kann in Software und — unter bestimmten Einschränkungen — auch durch High-Level-Synthese in einem Hardware-Koprozessor implementiert werden. Die explizit angegebenen Kommunikationskanäle beschreiben die nötigen Schnittstellen. Jedoch muß die Partitionierung vom Programmierer vorgegeben werden. Dieser prozeß-parallele Ansatz ist vor allem für irregulär kommunizierende Tasks in den oben erwähnten SP-Rechnern für Steuerungsaufgaben [WBO95] geeignet.

Im Gegensatz dazu wird ein SP-Rechner bei der Programmierung in dbC [GM95] als *SIMD-Rechner* mit Wirts- und Direkter-Nachbar-Kommunikation betrachtet. Der Typ einer vom Programmierer verwendeten Operation bestimmt, ob sie auf dem Wirt oder in den auf den FPGAs realisierten Prozessor-Elementen ausgeführt werden soll. Die Schaltungs-Synthese erzeugt dann ein Feld von Prozessor-Elementen, die genau die programmspezifischen Operationen ausführen können. Dieser datenparallele Ansatz ermöglicht eine effektive Nutzung der Hardware, ist aber nur für SIMD- und systolische Algorithmen geeignet. Beispielsweise wurde eine Version des Genom-Vergleichers in dbC implementiert.

Schließlich implementieren Iseli et al. [IS93] einen variablen *superskalaren Prozessor*. Dabei sind die meisten Teile des Prozessors fest. Nur die funktionalen Einheiten werden programmspezifisch synthetisiert. Dann erzeugt ein konventioneller Übersetzer Code für den Prozessor. Allerdings müssen die Operatoren, die in dem Prozessor realisiert werden sollen, manuell ausgewählt werden.

4.2 Automatische Hardware/Software-Partitionierung

Bei den oben vorgestellten kombinierten Programmiermethoden wird die Partitionierung in Hardware- und Software-Anteile entweder durch den Entwickler vorgegeben oder durch die Sprachkonstrukte selbst festgelegt. Eine automatische Partitionierung wäre aber sinnvoll, um dem Entwickler diese Aufgabe, für die Kenntnisse über die zur Verfügung stehende Hardware nötig sind, abzunehmen. Außerdem würde sie auch eine automatische Portierung beim Wechsel auf einen anderen SP-Rechner oder bei einer Umkonfigurierung der Hardware ermöglichen.

Solche automatischen Partitionierungs-Verfahren wurden bereits für eingebettete Systeme im Hardware/Software-Codesign entwickelt. Ihr Erfolg beruht auf der Erfahrung, daß die meisten Programme einen Großteil ihrer Rechenzeit in wenigen

inneren Schleifen verbrauchen. Dieser iterative Kern der Anwendung ist sehr regulär (im in Abschnitt 3.4.2 genannten Sinn) und eignet sich daher — wenn er nicht zu komplizierte Operationen enthält -- für eine Hardware-Beschleunigung. Der Rest des Programms besteht aus selten ausgeführten, irregulären Instruktionen, die besser in Software abgearbeitet werden. Die verschiedenen Ansätze unterscheiden sich nun in der Methode, wie geeignete Hardware-Koprozessoren identifiziert werden. Dabei muß der Nutzen durch die erwartete Beschleunigung immer im Verhältnis zu den entstehenden Kosten für Konfigurierung und Kommunikation betrachtet werden.

Das Partitionierungsziel bei eingebetteten Systemen ist, vorgegebene Leistungsanforderungen mit minimalen Kosten zu erreichen. Da die meisten Systeme sowieso einen Mikroprozessor oder -controller (zur Ausführung des Software-Anteils) enthalten, soll die Funktionalität so aufgeteilt werden, daß ein minimaler Bedarf an zusätzlicher Hardware (ASICs oder FPGAs) entsteht. Ein Beispiel für eine solche Partitionierung finden wir im COSYMA-System [EHB93]. Eingabesprache ist eine um Zeitbedingungen erweiterte C-Variante. COSYMA erkennt durch Profiling automatisch laufzeitkritische Teile des C-Programms und bestimmt eine Partitionierung, die die vorgegebenen Zeitbedingungen einhält. Eine iterative Optimierung durch *Simulated Annealing* bestimmt eine Lösung mit möglichst geringem Hardware-Bedarf. Für die ausgewählten Programmteile werden mit einem High-Level-Synthese-System Hardware-Koprozessoren generiert, die dann — ähnlich wie in PRISM und PRISM-II — vom in Software verbleibenden Rest des Programms aufgerufen werden.

Bei derartigen Verfahren dauert die genaue Schätzung des Hardware-Bedarfs für die vielen betrachteten Partitionierungen sehr lange. Sie kann jedoch durch die in [VG95] vorgestellte, während der Partitionierung durchgeführte *inkrementelle Hardware-Schätzung* wesentlich beschleunigt werden.

Bei der Programmierung von SP-Rechnern ist das Partitionierungsziel jedoch, die höchste Beschleunigung für eine Anwendung bei fest vorgegebenen Hardware-Ressourcen (des jeweiligen SP-Rechners) zu erreichen. Jantsch et al. [JEO+94b, JEO+94a] verwenden ebenfalls C als Eingabesprache und schränken den Suchraum für die Partitionierung ein, indem zuerst Hardware-Kandidaten (innere Schleifen und Funktionen) definiert und dann diejenigen ausgewählt werden, die die größte Beschleunigung erwarten lassen. Das in [Wei95] vorgeschlagene Partitionierungsverfahren baut auf diesen Verfahren auf, berücksichtigt aber die Kommunikationskosten zwischen Wirt und Koprozessor-Karte genauer und verwendet ganzzahlige lineare Optimierung (siehe Anhang B). Beide Verfahren bauen jedoch auf durch High-Level-Synthese erzeugten sequentiellen Koprozessoren auf und erreichen daher wie bereits mehrfach erwähnt — nur in wenigen Fällen eine für SP-Rechner ausreichende Beschleunigung. Solche Koprozessoren sind eher für den Einsatz in Massenprodukten geeignet, in denen statt FPGAs schnellere ASICs und möglichst billige (und daher auch langsamere) Prozessoren oder Controller verwendet werden.

Das von Hartenstein et al. [HBJ+96] vorgeschlagene Verfahren für Xputer-Systeme verwendet wie COSYMA Profiling-Daten und Simulated Annealing zur Bestimmung einer Partitionierung. Dabei wird ebenfalls die Beschleunigung maximiert.

Einige Verfahren teilen den Hardware-Anteil auch in verschiedene Konfigurationen auf, zwischen denen zur Laufzeit *dynamisch rekonfiguriert* wird. Da die Konfigurierung heutiger FPGAs recht lange dauert, lohnt sich eine Umkonfigurierung aber nur, wenn jede Phase lange genug dauert. Bisher bekannte Ansätze setzen daher eine manuelle temporale Partitionierung [HW95] voraus. Das System wird in wenige Phasen, zwischen denen zur Laufzeit *immer* umkonfiguriert wird, aufgeteilt.

Da die von unserem Übersetzer synthetisierten Koprozessoren (siehe Kapitel 5) besondere Randbedingungen an die Partitionierung stellen, wurde für sie ein neues Partitionierungsverfahren entwickelt, das in Kapitel 6 vorgestellt wird. Es kombiniert *dynamische Rekonfigurierung* mit *dynamischer Laufzeit-Auswahl.*

4.3 Pipeline-Synthese und -Optimierung

Da Pipelines für eine effiziente SP-Rechner-Programmierung sehr wichtig sind (siehe Abschnitt 3.4.2) und auch unsere Koprozessor-Synthese auf Pipeline-Schaltungen basiert (siehe Kapitel 5), werden wir im nächsten Abschnitt zunächst deren Grundlagen kurz erörtern, bevor die weiteren Abschnitte Synthese- und Optimierungsverfahren für Pipelines vorstellen.

4.3.1 Grundlagen der Pipeline-Verarbeitung

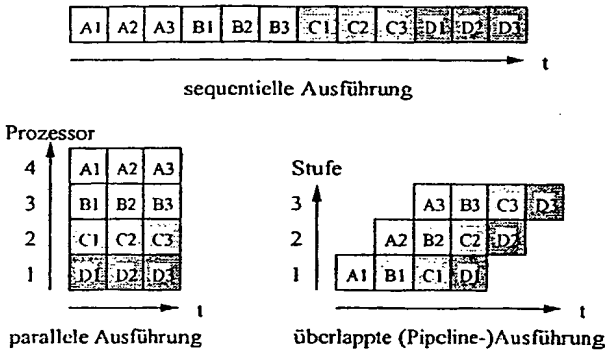


Abbildung 4.1: Pipeline-Prinzip

Die Pipeline-Verarbeitung ist ein weiteres Ausführungsmodell neben der sequentiellen und der parallelen Ausführung, das nicht nur im Hardware-Entwurf, sondern auch in vielen anderen Bereichen der Informatik eingesetzt wird. Es wird von dem deutschen Begriff "Fließband-Verarbeitung" eigentlich besser getroffen: Die Bearbeitungsschritte vieler gleichartiger Aufträge werden überlappt ausgeführt. In jeder Pipeline-Stufe wird immer derselbe Schritt durchgeführt, und jeder Auftrag

durchläuft alle Stufen in der gleichen Reihenfolge. Abbildung 4.1 stellt die sequentielle, parallele und überlappte Ausführung von vier Aufträgen (A bis D) mit je drei Bearbeitungsschritten einander gegenüber.

Die gesamte Bearbeitungszeit der Aufträge ist (wegen des Füllens und Leerlaufens der Pipeline) etwas länger als bei der parallelen Ausführung, aber wesentlich kürzer als bei der sequentiellen Ausführung. Denn nach dem Füllen sind immer alle Pipeline-Stufen aktiv; der Parallelitätsgrad ist (asymptotisch) gleich der Anzahl der Stufen. Dieser Grad wird bei der Verarbeitung großer Datenströme in Hardware-Pipelines, z. B. in einem Signalverarbeitungs-ASIC, nahezu erreicht. Können jedoch *Abhängigkeiten* zwischen aufeinanderfolgenden Aufträgen (d. h. Berechnungen) auftreten, dürfen sie nicht überlappen, und der theoretisch mögliche Parallelitätsgrad einer Pipeline wird kaum erreicht. Dies ist etwa beim *Befehlspipelining* in einem Prozessor der Fall, bei dem die Phasen der Befehlsausführung überlappt werden. Denn in gewöhnlichem Maschinencode führen Abhängigkeiten häufig zu einer Unterbrechung des Pipeline-Stromes. Außerdem benötigen nicht alle Befehle alle Pipeline-Stufen zur Ausführung, was in unterschiedlichen Ausführungszeiten resultiert. Diese Konflikte müssen in der Hardware erkannt oder von der Software (etwa durch zusätzlich eingefügte "leere" Befehle (NOPs) im Maschinencode) verhindert werden. Bei superskalaren Prozessoren kommt noch die Verwaltung verschiedener funktionaler Einheiten mit unterschiedlichen Ausführungszeiten hinzu, was den Einsatz komplizierter Verfahren zur Befehlsanordnung erfordert, um eine effektive Nutzung der Pipeline zu erreichen.

Außer in Hardware-Pipelines und beim Befehlspipelining wird das Pipeline-Prinzip auch in *Vektorrechnern* ausgenützt. Auf diese gehen wir in Abschnitt 4.4 näher ein.

4.3.2 Synthese aus daten-parallelen Programmen

Gewöhnlich werden Hardware-Pipelines manuell entworfen oder aus Datenfluß-Darstellungen, die bereits die Struktur der Pipeline repräsentieren, generiert. Im Gegensatz zu diesen Verfahren verwendet die von Guccione für SP-Rechner entwickelte Methode [Gue95, GG93, GG95b] *daten-parallele C*, also eine höhere Programmiersprache, zur Pipeline-Spezifikation. Die Programme dürfen jedoch keine Schleifen, sondern nur bedingte Anweisungen enthalten.

Zur Schaltungs-Synthese wird aus dem Eingabe-Programm zunächst ein Datenfluß-Graph generiert, indem für jede Anweisung ein Operator erzeugt wird (direkte Übersetzung). Dazu werden die Datenabhängigkeiten der Anweisungen analysiert, um die Verknüpfung der Operatoren zu bestimmen. Bedingte Anweisungen werden wie bei der Kontrollfluß-Datenfluß-Transformation aus Abschnitt 3.2.1 durch Multiplexer realisiert. Da alle Anweisungen des Programms auf Vektoren[2] definiert sind, können sie in einer Pipeline ausgeführt werden. So können auch zusätzliche *Präfix-* oder

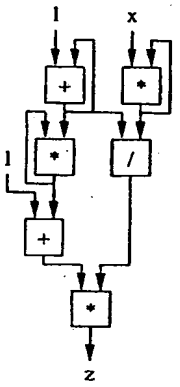
[2] Hier wird der Begriff *Vektor* als Synonym für *eindimensionales Feld* verwendet. In Abschnitt 3.2.1 fassen wir ihn jedoch enger.

Scan-Operatoren, z. B. `ADD-SCAN` oder `MIN-SCAN` verwendet werden, die für jeden Anfangs-Teilvektor ein kumuliertes Ergebnis berechnen. Denn diese Operatoren lassen sich in Pipelines leicht realisieren, indem das jeweils aktuelle Teilergebnis in einem Register gespeichert und zur Weiterverarbeitung im nächsten Pipeline-Takt rückgekoppelt wird. Aus einem Datenfluß-Graphen wird schließlich eine Netzliste generiert, indem die Operatoren mit Elementen einer Modul-Bibliothek instanziert werden. Abbildung 4.2 zeigt ein Beispiel für ein Programm in daten-parallelem C und den daraus erzeugten Datenfluß-Graphen.[3]

```
float y[SIZE];
float z[SIZE];

y = 1;              /* [1, 1, 1, 1, 1,...] */
y = add-scan(y);    /* [1, 2, 3, 4, 5,...] */
z = x;             /* [x, x, x, x, x,...] */
z = mult-scan(z); /* [x, x^2, x^3, x^4,...] */
z = y / z;       /* [1/x, 2/x^2, 3/x^3,...] */
y = mult-scan(y);   /* [1, 2, 6, 24,...] */
y = y + 1;          /* [2, 3, 7, 25,...] */
z = y * z;            /* [2/x, 6/x^2,...] */
```

(a)



(b)

Abbildung 4.2: Daten-paralleles C-Programm (a) und Datenfluß-Graph (b)

Die Pipeline-Ausführung hat mehrere Vorteile gegenüber einem sequentiellen Koprozessor, der dieselbe Funktion ausführt: Die Operatoren werden parallel ausgeführt, und die Schaltung benötigt durch Wiederverwendung von Eingabewerten und Zwischenergebnissen eine geringere E/A-Bandbreite. Und im Vergleich zu einer SIMD-Architektur werden für N-fache Parallelität nicht N universelle ALUs, sondern nur N feste Operatoren für die N Stufen der Pipeline gebraucht. Da nur die in jedem Schritt tatsächlich benötigten Operatoren instanziert werden müssen, werden die begrenzten Hardware-Ressourcen einer RE sehr gut ausgenutzt.

Allerdings hat das Verfahren auch einige Nachteile: Die Scan-Operatoren wurden speziell für die Pipeline-Synthese entworfen und müssen von einem Entwickler neu erlernt werden. Außerdem kann mit dieser Methode nur der Hardware-Anteil einer Anwendung spezifiziert und generiert werden, während die Ansteuerung der Pipelines und die Kopplung mit dem Wirt manuell implementiert werden muß. Und die Programme können nicht direkt auf dem Wirt getestet werden. Deshalb ist auch diese Methode für Software-Entwickler nicht einfach und allgemein genug einsetzbar.

[3] Man beachte, daß in der Arbeit von Guccione die unrealistische Annahme getroffen wird, daß auch Gleitkomma-Operatoren in rekonfigurierbarer Hardware implementiert werden können.

4.3.3 Pipelining in der High-Level-Synthese

Wie in Abschnitt 3.2.1 bereits erwähnt, wurden für die High-Level-Synthese erweiterte Ablaufplanungs-Verfahren entwickelt, die die Synthese von Pipelines für Schleifen ermöglichen. Diese Verfahren verfolgen eine ähnliche Zielsetzung wie unsere in Kapitel 5 vorgestellte Synthese-Methode.

Holtmann [Hol95, HE95] stellt ein Verfahren zum Pipelining allgemeiner WHILE- und FOR-Schleifen mit datenabhängiger Iterationszahl vor. Der Schwerpunkt dieses Ansatzes liegt auf der Optimierung von Schleifen mit vielen bedingten Anweisungen durch spekulative Ausführung: Die Pipeline wird für den wahrscheinlichsten Fall optimiert, während für die seltener auftretenden anderen Fälle eine längere Ausführungszeit durch eventuell notwendige Korrekturen in Kauf genommen wird.

Bei diesem Verfahren wird jedoch keine genaue Analyse der Abhängigkeiten von Feldzugriffen (wie in vektorisierenden Übersetzern für Vektorrechner) vorgenommen, so daß Schreib- und Lesezugriffe auf ein Feld in einer Schleife immer als voneinander abhängig betrachtet werden müssen. So können insbesondere für FOR-Schleifen mit vielen Feldzugriffen keine Pipelines synthetisiert werden.

4.3.4 Register-Verteilung und Optimierung

In einer Hardware-Pipeline (ASIC oder FPGA) muß die Signalverzögerung klein und folglich die erreichbare Taktfrequenz hoch sein, um einen hohen Durchsatz zu erreichen. Deshalb werden Zwischenergebnisse in Registern gespeichert, um die Operatoren zu entkoppeln und so parallele Berechnungen zu ermöglichen. Die Register müssen jedoch richtig verteilt sein, um die Korrektheit der Pipeline zu gewährleisten. Dies wird durch den im nächsten Abschnitt beschriebenen *Register-Ausgleich* erreicht. Danach werden die Verfahren *Retiming* und *Pipelining* vorgestellt, mit denen der Durchsatz einer bereits korrekten Pipeline erhöht werden kann. Zuletzt erörtern wir Möglichkeiten, den Bedarf an kombinatorischer Logik in einer Pipeline zu reduzieren.

Register-Ausgleich

Ein *Register-Ausgleich*[4] [Gao89, Guc95] einer Pipeline ist notwendig, wenn die Zahl der Register auf verschiedenen Pfaden zwischen einem Eingang und einem Ausgang der Pipeline unterschiedlich ist, die Schaltung also nicht korrekt synchronisiert ist. Unter der Annahme, daß jeder Operator bereits ein Register zur Speicherung seines Ausgabewertes enthält, ist dies in dem Beispiel-Graphen aus Abbildung 4.2(b) der Fall: Die Pfade durch den Divisions-Operator enthalten drei Register, die übrigen vier. Deshalb müssen weitere Register in die Pipeline eingefügt werden, damit die richtigen Werte zum richtigen Zeitpunkt an den Operatoren anliegen. Dies kann

[4] engl. register balancing

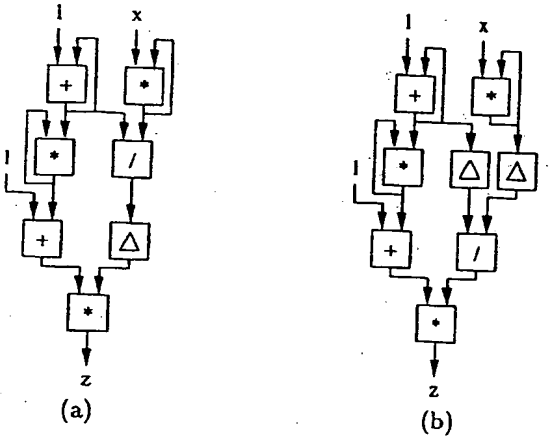


Abbildung 4.3: Ausgeglichene Datenfluß-Graphen

aber an verschiedenen Stellen der Pipeline geschehen — in unserem Beispiel etwa vor oder hinter dem Divisions-Operator (siehe Abbildung 4.3). Hier ist offensichtlich, daß beim Einfügen hinter dem Operator nur ein Register, davor aber zwei notwendig sind. Lösung (a) beansprucht also weniger Hardware. Im allgemeinen Fall hat die Plazierung eines Registers jedoch Auswirkungen auf andere Teile der Pipeline, so daß eine globale Register-Minimierung notwendig wird.

Dieses Optimierungsproblem tritt auch bei *statischen Datenflußrechnern* auf: Azyklische Pipelines müssen ausgeglichen werden, um den Durchsatz bei iterativen Berechnungen zu erhöhen. Gao hat dafür einen auf *ganzzahliger linearer Optimierung* (integer linear programming, ILP) [Len90] basierenden Lösungsansatz vorgeschlagen [Gao89]. Die Grundidee besteht darin, für jeden Operator die *Latenz* — also die Zahl der Pipeline-Stufen zwischen einem Eingang und dem Operator — zu berechnen. Werden dann zwischen zwei Operatoren so viele Register eingefügt, wie die Differenz der Latenzen erfordert, ist die Pipeline automatisch ausgeglichen. Die Latenzen werden nun so festgelegt, daß die Gesamtzahl der Register minimal ist. Die Zeitkomplexität des Verfahrens ist polynomiell in der Größe der Pipeline. Um dieses Verfahren auf Hardware-Pipelines anzuwenden, müßten jedoch die Taktfrequenz und die Zahl der internen Register der Operatoren künstlich festgelegt werden. Denn diese Werte sind bei statischen Datenfluß-Rechnern fest vorgegeben.

Retiming und Pipelining

Retiming ist der Sammelbegriff für eine Reihe von Verfahren zur Optimierung synchroner Schaltungen, die von Leiserson, Rose und Saxe [LRS83, LS83] entwickelt wurden. Alle Verfahren modellieren Schaltungen einheitlich als gerichtete Graphen und haben polynomielle Laufzeiten in der Größe der Graphen. Durch Verschieben

vorhandener Register über Operatoren hinweg wird die Verzögerung der Schaltung und die Zahl der Register verändert. Das an einem speziellen Knoten, dem *Schnittstellen-Knoten*, beobachtbare externe Verhalten einer Schaltung bleibt dabei unverändert. Beispielsweise können die beiden Graphen in Abbildung 4.3 durch Retiming ineinander überführt werden — ihr externes Verhalten ist identisch. Die Werte interner Register, die ja verschoben werden können, ändern sich aber. Folglich dürfen Schaltungen, bei denen interne Register direkt initialisiert oder ausgelesen werden, nicht durch Retiming verändert werden.

Die wichtigste Version des Retiming minimiert die Verzögerung einer Schaltung, maximiert also ihren Durchsatz. Mit einer anderen Version kann aber auch die Zahl der insgesamt benötigten Register bei vorgegebener Verzögerung minimiert werden.

Pipelining kann als Sonderfall des Retiming mit mehreren Schnittstellen-Knoten betrachtet werden. Auch hier wird der für die Leistungsfähigkeit entscheidende Durchsatz einer synchronen Schaltung erhöht, und die extern beobachtbare Funktionalität bleibt erhalten. Im Gegensatz zum einfachen Retiming darf jedoch durch das Einfügen weiterer Pipeline-Stufen die Synchronisation (also das Zeitverhalten) der verschiedenen Schnittstellen-Knoten untereinander verändert werden: Die Zahl der Takte zwischen dem Beginn und dem Ende einer einzelnen Berechnung — also die Latenz der Pipeline — darf sich vergrößern.

Beim Pipelining müssen auf allen azyklischen Pfaden zwischen zwei Schnittstellen-Knoten (einem Eingang und einem Ausgang der Schaltung) gleich viele Register eingefügt werden. Nur dann liegen zusammengehörende Werte weiterhin zum gleichen Zeitpunkt an den Operatoren an. Da beliebig viele Stufen eingefügt werden können, ist der Durchsatz azyklischer Pipelines nur durch den langsamsten Operator begrenzt. Dessen Durchsatz kann jedoch auch durch zusätzliche interne Pipeline-Stufen erhöht werden.

Enthält der Schaltungs-Graph jedoch Zyklen, so dürfen innerhalb derselben keine Register eingefügt werden, da dies die Funktionalität verändern würde. Die Zyklen beschränken also den Durchsatz einer solchen Pipeline. In Abbildung 4.3 kann beispielsweise nur der Durchsatz der nicht rückgekoppelten Operatoren durch zusätzliche interne Pipeline-Stufen erhöht werden.

[MNT+94] stellt ein ebenfalls auf Pipelining basiertes Verfahren zur nachträglichen Optimierung von FPGA-Schaltungen vor. Zur Synthese optimaler Pipelines sollte jedoch von vornherein das Pipelining mit dem Register-Ausgleich kombiniert werden. Deshalb erweitern wir in Abschnitt 5.4.2 das Verfahren von Cao so, daß es auch FPGA-Pipelines mit flexibler Operatorverzögerung ausgleicht und gleichzeitig durch Pipelining eine vorgegebene Verzögerung (d. h. Taktfrequenz) erreicht. Dabei wird die Zahl der zusätzlich eingefügten FPGA-Flipflops minimiert.

Minimierung kombinatorischer Logik

Der Bedarf an kombinatorischer Logik läßt sich bei Pipelines im allgemeinen nicht — wie bei der High-Level-Synthese — durch Mehrfachnutzung der Operatoren reduzieren, da alle Operatoren gleichzeitig aktiv sind. Die einzige Möglichkeit zur Minimierung besteht darin, die durch bedingte Anweisungen entstandenen alternativen Pfade im Datenfluß-Graphen zu optimieren. Denn von diesen Pfaden wird immer nur einer durch einen Multiplexer ausgewählt. Wird beispielsweise in einem Pfad $(X+Y)$ und im anderen Pfad $(X-Y)$ berechnet, so können der Addierer, der Subtrahierer und der Multiplexer durch einen kombinierten ADD/SUB-Operator ersetzt werden. Das Auswahlsignal des ursprünglichen Multiplexers bestimmt dann die Funktion des kombinierten Operators. Auf diesem Prinzip basierende, kompliziertere Verfahren zur Pipeline-Minimierung werden in dem PISYN-System [CW91, Kapitel 3] angewendet.

4.4 Parallelität durch Vektorisierung

Dieser Abschnitt erläutert zunächst die Architektur und die speziellen Befehle von Vektorrechnern, die Daten ebenfalls in Pipelines verarbeiten. Der Schwerpunkt des Abschnitts liegt auf der automatischen Vektorisierung sequentieller Programme für Vektorrechner.

4.4.1 Vektorrechner und Vektoranweisungen

In *Vektorrechnern* wird ein höherer Durchsatz erreicht, indem die gleichen arithmetischen Operationen auf einem Vektor von Daten ausgeführt werden (funktionale Pipelines). Dazu werden spezielle Vektorbefehle verwendet, die garantieren, daß keine Abhängigkeiten zwischen den Berechnungen bestehen. Durch direktes Verketten[5] der Vektoreinheiten lassen sich bei manchen Vektorrechnern tiefere Pipelines erreichen. Bei Multi-Vektor-Rechnern [HX88] können sie über Schieberegister sogar beliebig zu verzweigten Pipelines verkettet werden. Dadurch entstehen ähnliche Optimierungsprobleme wie bei Hardware-Pipelines: Da jeder Operator eine konstante Zahl interner Pipeline-Stufen hat, muß eine vorgegebene Pipeline durch Retiming so modifiziert werden, daß für jeden Operator auch genügend Pipeline-Stufen vorhanden sind. Weil dies nicht immer möglich ist, kann auch nicht immer in jedem Takt ein Ergebnis berechnet werden. So hängt der Durchsatz auch hier — trotz durch die Rechnerarchitektur fest vorgegebener Taktfrequenz — von Pipeline-Optimierungen ab. In [HX88] wird ein Pipeline-Optimierungsverfahren für Multi-Vektor-Rechner beschrieben.

Zur Programmierung von Vektorrechnern werden vor allem imperative Programmiersprachen verwendet, die um spezielle *Vektoranweisungen* erweitert wurden.

[5] engl. chaining

Diese Anweisungen wenden bekannte Operatoren auf Vektoren an und können direkt in die entsprechenden Maschinenbefehle des Vektorrechners übersetzt werden. Z. B. bezeichnet in FORTRAN90 `A(I:J:S)` alle Elemente des Vektors `A` von `I` bis `J` mit der Schrittweite[6] `S`. Wird `S` weggelassen, so gilt die Schrittweite 1. Eine Vektoraddition sieht dann beispielsweise so aus:

```
A(1:50) = B(1:50) + C(1:99:2)
```

Implizit stellt jede Vektoranweisung eine parallele Schleife für die Zuweisungen an die selektierten Vektorelemente dar. Über die Reihenfolge der Zuweisung wird keine Annahme gemacht, jedoch gilt die "fetch before store"-Semantik. Sie besagt, daß zuerst alle benötigten Werte geladen werden, bevor ein Ergebnis gespeichert wird.

Daneben gibt es noch spezielle *Reduktions-Operatoren*, die aus einem oder mehreren Vektoren einen skalaren Wert berechnen. Dies sind etwa das Maximum `MAXVAL(A)` und die Summe `SUM(A)` eines Vektors, oder das Skalarprodukt `DOTPRODUCT(A,B)` zweier Vektoren. Derartige Anweisungen werden auch in Programmiersprachen für SIMD-Rechner verwendet. Sie entsprechen den *Scan-Operatoren* des daten-parallelen C aus Abschnitt 4.3.2, liefern aber nur ein Ergebnis für den ganzen Vektor.

Auch die anderen daten-parallelen C-Anweisungen können als vereinfachte Vektoranweisungen betrachtet werden. Da Zugriffe mit einer Schrittweite größer eins durch einen speziellen STRIDE-Operator und eine Zuweisung an einen neuen Vektor (mit Schrittweite eins) realisiert werden müssen, definieren Zuweisungen an einen Vektor immer alle Elemente neu. Folglich ist eine Verkettung der Operatoren immer möglich, ohne daß eine genauere Analyse auf der Ebene einzelner Vektor-Elemente erfolgen muß. Nicht verkettete Vektoroperationen wären in einer Hardware-Pipeline auch nicht möglich, da in Registern nur ein Wort als Zwischenergebnis gespeichert werden kann, aber keine großen Vektorregister zur Zwischenspeicherung ganzer Vektoren zur Verfügung stehen.

4.4.2 Automatische Vektorisierung

Neben der direkten Verwendung der Vektoranweisungen wurden zur Vereinfachung der Programmierung und vor allem zur Wiederverwendung existierender Programme auch Übersetzer zur *automatischen Vektorisierung sequentieller Programme* entwickelt [AK87, ZC91, Wol96]. Sie ersetzen Schleifen, die Felder (Vektoren) bearbeiten, durch semantisch äquivalente, parallel ausführbare Vektoranweisungen. Diese Vektorisierung ist ein Spezialfall der Programm-Parallelisierung, die für Multiprozessor-Systeme eingesetzt wird. Automatische Methoden können jedoch nicht immer die in einem Programm implizit vorhandene Parallelität erkennen. Erst recht können sie keinen besseren, für einen Vektor- oder Parallelrechner geeigneteren Algorithmus erzeugen.

[6] engl. stride

In einem sequentiellen Programm ist eine lineare Reihenfolge der Befehlsausführung vorgegeben. Diese Reihenfolge ist normalerweise restriktiver als nötig, also überspezifiziert. Deshalb wird bei der Vektorisierung versucht, die durch Datenabhängigkeiten verursachten semantisch relevanten Reihenfolge-Einschränkungen zu ermitteln. Die anderen, irrelevanten Einschränkungen dürfen ignoriert werden, so daß die Reihenfolge der Befehle nur noch teilweise festgelegt ist. Unter Berücksichtigung dieser partiellen Ordnung werden die Schleifen des Programms dann so umgeordnet, daß sie in Vektoranweisungen umgewandelt werden können. Bei der Vektorisierung werden im einzelnen die folgenden Schritte durchgeführt:

Schleifen-Transformationen

Diese Transformationen auf Quellsprach-Ebene sind nicht unbedingt nötig, können aber die Ergebnisse der nachfolgenden Vektorisierung verbessern oder diese erst ermöglichen. Wir stellen hier zwei Transformationen vor, die im weiteren Verlauf dieser Arbeit benötigt werden:

Die *Schleifen-Aufteilung*[7] teilt das von der Schleifenvariable durchlaufene Intervall in einzelne Abschnitte (Blöcke) auf, so daß mehrere kleinere Schleifen enstehen. Sind alle Blöcke gleich groß, können sie in einer neuen, äußeren Schleife zusammengefaßt werden. Die Indizes im Rumpf der inneren Schleife müssen dann entsprechend angepaßt werden. Diese Transformation kann durchgeführt werden, um eine Schleife an die Länge der Vektorregister auf einer Maschine anzupassen oder um eine Abhängigkeit zu eliminieren. Eine Schleifen-Aufteilung ist immer zulässig, da die Ausführungsreihenfolge der Iterationen nicht geändert wird.

Im Gegensatz dazu werden beim *Schleifen-Austausch*[8] innere und äußere Schleifen vertauscht, die Reihenfolge ändert sich also. Wegen möglicher Abhängigkeiten zwischen den Anweisungen ist diese Transformation natürlich nicht immer zulässig. Deshalb müssen vor ihrer Anwendung bestimmte Bedingungen [ZC91] überprüft werden. Der Schleifen-Austausch wird beispielsweise verwendet, um möglichst lange Schleifen zu vektorisieren.

Normalisierung

Nach der Transformation werden die (eventuell geschachtelten) Schleifen durch die folgenden Schritte normalisiert:

- *Schleifenvariablen-Substitution:* Suche Variablen, die in jeder Iteration um einen konstanten Wert erhöht oder erniedrigt werden. Sind mehrere solche *Schleifenvariablen* (oder Induktionsvariablen) vorhanden, so ersetze sie durch eine Haupt-Schleifenvariable. (Dies ist bei FOR-Schleifen die im Schleifenkopf definierte Variable.)

[7] engl. loop sectioning, strip-mining
[8] engl. loop-interchange

- *Normalisierung der Schleifenvariablen:* Substituiere die Schleifenvariable durch eine neue Variable, die mit dem Wert 1 (oder 0) initialisiert und in jeder Iteration um 1 inkrementiert wird.
- *Ausdrucks-Faltung:* Ersetze im Schleifenrumpf Konstanten und Variablen, die Zwischenergebnisse speichern, durch ihre Werte.
- *Normalisierung der Indexausdrücke:* Vereinfache Feldzugriffe (falls möglich) in Ausdrücke, die nur linear von den Schleifenvariablen sowie von außerhalb der Schleife definierten Variablen abhängen.

Abhängigkeits- und Alias-Analyse

Als nächstes werden die Datenabhängigkeiten der Schleifen des Programms analysiert. Es gibt drei prinzipielle Abhängigkeits-Typen:

- *Echte Abhängigkeit:* Anweisung B hängt echt von Anweisung A ab, falls eine Variable (oder Speicherzelle) in A definiert (geschrieben) und — ohne zwischenzeitliche Undefinition — in B benutzt (gelesen) wird. Dabei muß B nach A ausgeführt werden.
- *Anti-Abhängigkeit:* Diese Abhängigkeit besteht, wenn durch Vertauschen von A und B eine echte Abhängigkeit entstehen würde.
- *Ausgabe-Abhängigkeit:* A und B sind ausgabe-abhängig, wenn beide dieselbe Variable (Speicherzelle) definieren. Eine Vertauschung der Anweisungen würde also zu unterschiedlichen Ausgaben führen.

Außerdem unterscheiden wir *schleifenunabhängige Abhängigkeiten*, die zwischen Anweisungen derselben Schleifeniteration auftreten, und *schleifengetragene Abhängigkeiten*, die zwischen verschiedenen Iterationen auftreten. Bei letzteren bezeichnet die *Distanz der Abhängigkeit* die Anzahl der Iterationen, die zwischen den beiden Anweisungen liegen. Bei geschachtelten Schleifen sprechen wir von *Distanzvektoren*.

Für die Vektorisierung müssen schleifengetragene Abhängigkeiten erkannt werden. Dazu ist eine *Alias-Analyse* der Feldzugriffe nötig. Deshalb definieren wir zunächst den Begriff *Alias*: Ein Alias liegt vor, wenn zwei verschiedene Ausdrücke in einem Programm zur Laufzeit dieselbe Speicheradresse referenzieren. Da diese Eigenschaft im allgemeinen zur Übersetzungszeit nicht bestimmt werden kann, muß sie approximiert werden. Wir unterscheiden zwei Fälle:

- Ein *MUST-Alias* liegt vor, wenn zwei Ausdrücke zur Laufzeit *immer* Aliase voneinander sind.
- Ein *MAY-Alias* liegt vor, wenn zwei Ausdrücke zur Laufzeit nur *manchmal* Aliase voneinander sind.

Es gibt mehrere Ursachen für Aliase: Zunächst können verschiedene Zeiger dieselben Adressen enthalten. Außerdem können verschiedene Referenzparameter dieselbe Variable bezeichnen. Solche Aliase können durch eine interprozedurale Datenfluß-Analyse erkannt werden. Schließlich sind alle Zugriffe auf ein Feld mögliche Aliase, da verschiedene Indexausdrücke denselben Wert haben können.

Für die Abhängigkeits-Analyse sind die MUST- und die MAY-Aliase der in den Anweisungen definierten und benutzten Feldelemente relevant. Um sie vergleichen zu können, wurden die Indexausdrücke normalisiert. Für die Analyse müssen im Falle linearer Indexausdrücke *lineare diophantische Gleichungen* gelöst werden. Leider ist dies im allgemeinen nicht exakt möglich, da nur notwendige Voraussetzungen für eine Lösung überprüft werden können, z. B. mit dem *ggT-Test* und dem *Banerjee-Test*. Im Zweifelsfall muß die konservative Annahme getroffen werden, daß eine Lösung und damit eine Abhängigkeit vorhanden ist. Dies kann eine eigentlich zulässige Vektorisierung verhindern. Lediglich für den einfachen Fall, daß nur eine Schleifenvariable in den Ausdrücken vorkommt (z. B. bei nicht geschachtelten Schleifen) gibt es den exakten, aber aufwendigen *Separabilitäts-Test* [ZC91].

Erzeugung der Vektoranweisungen

Schließlich werden die Anweisungen des Schleifenrumpfes gegebenenfalls umgeordnet und auf möglichst viele einzelne Schleifen verteilt. Bei zyklischen Abhängigkeiten zwischen den Anweisungen ist dies jedoch nicht möglich. Enthält eine resultierende Schleife aber nur eine Anweisung, die nicht von sich selbst abhängt, so kann die ganze Schleife durch eine Vektoranweisung ersetzt werden. Manche Übersetzer können in bestimmten Fällen sogar automatisch Reduktions-Operationen generieren.

Jedoch lassen sich weder mit den bekannten Vektorisierungsverfahren noch durch direkte daten-parallele Programmierung mit Präfix-Operatoren allgemeine Pipelines erzeugen, die beliebige Rückkopplungen über mehrere Operator-Stufen enthalten. Dies ist jedoch mit dem im nächsten Kapitel vorgestellten, erweiterten Vektorisierungsverfahren möglich.

Kapitel 5

Vektorisierung und Pipeline-Synthese

Mit diesem Kapitel beginnt der Kern dieser Arbeit. Es stellt neue Methoden zur Synthese von Pipeline-Schaltungen aus Schleifen eines imperativen Programmes vor. Sie lösen die in Abschnitt 2.2 genannten Teilprobleme Kandidaten-Auswahl und Koprozessor-Synthese und können als Komponenten des in Abbildung 2.9 (Seite 10) vorgestellten Gesamtsystems eingesetzt werden. Die automatische Integration der Pipelines in eine Anwendung wird in Kapitel 6 vorgestellt. Ebenso behandeln wir dort die Partitionierung, also die Frage, für welche der prinzipiell vektorisierbaren Schleifen sich die Auslagerung auf die RE zur Programmbeschleunigung überhaupt lohnt.

Zunächst betrachten wir ein einfaches Verfahren zur Koprozessor-Synthese aus sequentiellen Programmiersprachen. Es zeigt, welche allgemeinen Voraussetzungen erfüllt sein müssen, um aus Programmteilen Hardware zu generieren, ist jedoch auf Programmteile ohne Schleifen beschränkt. Anschließend erweitert Abschnitt 5.2 das Verfahren auf FOR-Schleifen ohne echte schleifengetragene Abhängigkeiten. Dabei wird die Vektorisierung sequentieller Sprachen (vgl. Abschnitt 4.4.2) mit High-Level-Synthese-Methoden (vgl. Abschnitt 3.2.1) kombiniert, um effiziente Pipeline-Schaltungen, die die Hardware-Operatoren parallel nutzen, zu synthetisieren. Diese Pipeline-Synthese wird in Abschnitt 5.3 auf die Klasse der Schleifen mit regulären Abhängigkeiten erweitert. Danach entwickeln wir ein formales Verfahren zur globalen Registeroptimierung, das die Methoden aus Abschnitt 4.3.4 für FPGA-Pipelines anpaßt und erweitert. Schließlich zeigt Abschnitt 5.5, wie die einzelnen Methoden in einem Übersetzer kombiniert werden können, und Abschnitt 5.6 faßt die Ergebnisse zusammen und bewertet sie.

Die Hauptideen dieses Kapitels wurden bereits in [Wei96a, Wei96c, Wei97a, Wei97b] veröffentlicht.

5.1 Koprozessoren für schleifenfreie Programmteile

Dieser Abschnitt legt zunächst die für alle Beispiele verwendete Eingabesprache fest und identifiziert Voraussetzungen für die Koprozessor-Synthese aus sequentiellen Programmiersprachen. Denn vor der Synthese muß getestet werden, ob ein Programmteil überhaupt ein zulässiger Hardware-Kandidat ist. Dabei erfordern die Datentypen (z. B. Felder) und Programmkonstrukte (z. B. Prozeduren) höherer Programmiersprachen auch weitergehende Analysen wie die Alias-Analyse.

Danach wird eine Synthesemethode für einfache Koprozessoren, die nur kombinatorische Logik enthalten, vorgestellt. Diese evaluieren alle Anweisungen des Kandidaten durch einen Datenfluß-Graphen und sind deshalb auf Programmteile ohne Schleifen beschränkt. Sie können aber auch als Grundlage für die erweiterten Methoden (Abschnitte 5.2 und 5.3), die Kandidaten mit Schleifen behandeln, eingesetzt werden.

Wir beschäftigen uns nicht mit der logischen Optimierung der kombinatorischen Koprozessoren, da dies bereits im PRISM-System (Abschnitt 4.1) für die speziellen kombinatorischen Funktionen betrachtet wurde (vgl. Abschnitt 3.4.2). Allerdings birgt die iterative Berechnung spezieller kombinatorischer Funktionen — also Schleifen, in denen diese Funktionen vorkommen — ein besonderes Beschleunigungspotential. Wir werden diese Möglichkeit in Abschnitt 9.1 diskutieren.

5.1.1 Eingabesprache

Wie bereits in Kapitel 2 erläutert, wollen wir aus einer sequentiellen, imperativen Programmiersprache Koprozessoren generieren, damit auch Software-Entwickler, die keine Erfahrung im Hardware-Entwurf haben, SP-Rechner einsetzen können. Da der im Rahmen dieser Arbeit entwickelte Übersetzer-Prototyp (Kapitel 7) MODULA-2 verwendet, formulieren wir alle Beispielprogramme in MODULA-Syntax. Es könnte ebensogut eine andere imperative Sprache wie FORTRAN oder C verwendet werden, im Hardware-Anteil einer Anwendung nur Anweisungen auf elementaren Datenen und Feldern sowie bedingte Anweisungen und Schleifen auftreten dürfen, die allen Sprachen vorkommen. Einzige Ausnahme ist die Verwendung der Unterbehstypen von MODULA-2, um Daten beliebiger Wortlänge zu spezifizieren, etwa .63] für eine vorzeichenlose 6-Bit-Ganzzahl. Dies ist in vielen anderen Sprachen t direkt möglich, aber für flächen-effiziente Schaltungen in vielen Fällen notwenFür effiziente Koprozessoren wären auch Festkommatypen und -operationen cich, die aber bei den meisten modernen Programmiersprachen — mit Ausne von ADA — leider fehlen.

5.1.2 Hardware-Kandidaten

Ein Hardware-Kandidat darf keine nicht synthetisierbaren Funktionen enthalten. Dies sind aus prinzipiellen Gründen externe Bibliotheks- oder Betriebssystem-Aufrufe. Aus praktischen Gründen können auch keine Gleitkomma-Operationen durch Hardware-Operatoren realisiert werden, da diese zu groß sind. Außerdem können bei der einfachen Synthesemethode keine Schleifen und Funktionsaufrufe behandelt werden. Ein Kandidat darf also nur Zuweisungen und bedingte Anweisungen enthalten.[1]

```
CONST MAXITER = 10;
TYPE SCARD    = [0..65535]; (* 16-bit cardinal *)
VAR  S:       [0..1];
     X:       ARRAY[0..1],[0..N] OF SCARD;
...
S := 0;       (* X[0] is source array *)
FOR ITER := 1 TO MAXITER DO (* L1 *)
  FOR I := 3 TO N DO        (* L2 *)
    X[1-S,I] := 0;
    FOR K := 0 TO 3 DO      (* L3 *)
      X[1-S,I] := X[1-S,I] + X[S,I-K] DIV 4;
    END;
  END;
  S := 1 - S;
END
```

Abbildung 5.1: Programmbeispiel FIR-Filter

Wir wollen die Koprozessor-Synthese an einem durchgehenden Programmbeispiel, dem in Abbildung 5.1 gezeigten vierstufigen FIR-Filter, erläutern.[2] Es filtert dieselben Daten mehrmals und kopiert sie dabei zwischen den eindimensionalen Feldern X[0] und X[1] hin und her. Aus Abbildung 5.1 geht hervor, daß es in diesem Programm keinen Kandidaten gibt, der mehr als eine Zuweisung enthält. Deshalb sind die möglichen Koprozessoren sehr klein und können auch keine Berechnungen in der Hardware parallel ausführen. Diese Situation kann in vielen Fällen durch das vollständige Ausrollen kleinerer innerer Schleifen und die Expansion nicht-rekursiver Funktionen geändert werden. Wird etwa in unserem Beispielprogramm die innere Schleife L3 ausgerollt, so ist der ganze Rumpf der resultierenden neuen Schleife L2 (Abbildung 5.2) ein Kandidat.

[1] Die in MODULA definierten Zuweisungen ganzer Felder behandeln wir, indem wir sie durch einzelne Zuweisungen der Feldelemente ersetzen. Zugriffe auf Verbund-Datentypen werden hier nicht näher betrachtet, sind aber prinzipiell zulässig.

[2] Diese Anwendungsklasse wird in Abschnitt 8.2.1 näher vorgestellt.

```
...
S := 0;       (* X[0] is source array *)
FOR ITER := 1 TO MAXITER DO (* L1 *)
  FOR I := 3 TO N DO        (* L2 *)
    X[1-S,I] := 0;
    X[1-S,I] := X[1-S,I] + X[S,I] DIV 4;
    X[1-S,I] := X[1-S,I] + X[S,I-1] DIV 4;
    X[1-S,I] := X[1-S,I] + X[S,I-2] DIV 4;
    X[1-S,I] := X[1-S,I] + X[S,I-3] DIV 4;
  END;
  S := 1 - S;
END
```

Abbildung 5.2: FIR-Filter mit ausgerollter Schleife L3

5.1.3 Alias-Analyse

Vor der Koprozessor-Synthese müssen mögliche Aliase in den Kandidaten analysiert werden. Denn bei der Konstruktion eines Datenfluß-Graphen werden berechnete Werte direkt mit den Operatoren, die sie weiterverwenden, verknüpft. Die Identität der Variablen und Feldelemente, die im Kandidaten definiert werden, muß also durch eine exakte Alias-Analyse bestimmt werden, damit die abhängigen Operatoren die richtigen Werte weiterverwenden können. Denn wenn ein Alias zweier Variablen vorliegt, müssen sie als dieselbe Variable betrachtet werden, und es ergibt sich ein anderer Datenfluß-Graph als ohne Alias. MAY-Aliase, die zur Laufzeit nur manchmal auftreten, verhindern also die Konstruktion eines eindeutigen Datenfluß-Graphen. Ein Alias muß entweder *immer* oder *niemals* auftreten.

Dagegen können Variablen, die im Kandidaten nur benutzt, aber nicht definiert werden, immer als unabhängige Eingaben betrachtet werden. Für sie ist also keine Alias-Analyse nötig.

Von den möglichen Ursachen für Aliase (vgl. Abschnitt 4.4.2) lassen wir Zeiger-Operationen in Hardware-Kandidaten gar nicht zu.[3] Durch Referenzparameter verursachte Aliase werden durch bekannte interprozedurale Datenfluß-Analyse-Verfahren erkannt und werden deshalb hier nicht weiter betrachtet. Jedoch müssen Aliase verschiedener Feldzugriffe genauer analysiert werden. Um deren Indexausdrücke überhaupt vergleichen zu können, müssen sie zunächst normalisiert werden.

[3] Da die später betrachtete Ziel-Hardware Zeiger nicht dereferenzieren kann, müssen wir Zeiger-Operationen sowieso als nicht synthetisierbar betrachten.

›rmalisierung der Indexausdrücke

ersten Schritt der Normalisierung werden in allen Indexausdrücken Zwischen-
ebnisse, d. h. Variablen, die in dem Kandidaten definiert wurden, durch ihre
finitionen ersetzt. Ergibt sich dadurch eine indirekte Adressierung, können wir
: Aliase nicht exakt bestimmen und müssen MAY-Aliase annehmen. Wir können
nn also keinen Koprozessor für den Kandidaten erzeugen. Andernfalls werden die
sdrücke im zweiten Schritt in möglichst lineare Ausdrücke der außerhalb des Kan-
laten definierten Variablen umgeformt. Unser Beispielprogramm (Abbildung 5.2)
thält keine indirekte Adressierung, und alle Indexausdrücke sind bereits linear.

›rgleich der Indexausdrücke

ır eigentlichen Alias-Analyse müssen die möglichen Werte der Indizes aller defi-
›rten Felder verglichen werden: Um MAY-Aliase auszuschließen, dürfen zwei ver-
›iedene Indexausdrücke zur Laufzeit nie denselben Wert annehmen, also dasselbe
ldelement referenzieren. Da in unserem Kandidaten in Abbildung 5.2 nur das
ldelement `X[1-S,I]` definiert wird, ist hier keine Analyse nötig.

ie Analyse entfällt ebenfalls, wenn alle Indizes Konstanten sind. Andernfalls
üssen die möglichen Werte der außerhalb des Kandidaten definierten Variablen
›trachtet werden. Dafür gibt es verschiedene Möglichkeiten:

- Haben die Variablen nur einen kleinen Wertebereich (etwa bei Unterbereichstypen), dann kann die Gleichheit der Ausdrücke durch vollständige Aufzählung aller Belegungen getestet werden. In Abbildung 5.2 ist dies für die Indexausdrücke `S` und `1-S` des Feldes `X` der Fall: Sie nehmen für beide zulässigen Belegungen von `S` (0 und 1) unterschiedliche Werte an. Ein Alias ist also ausgeschlossen.
- Unterscheiden sich die Ausdrücke durch konstante Summanden, so sind sie auch immer verschieden. Dies ist etwa für die Indizes `I`, `I-1`, `I-2` und `I-3` in Abbildung 5.2 der Fall.
- In manchen Fällen können auch durch eine *abstrakte Interpretation* der Zuweisungen Informationen über die Werte zur Laufzeit gewonnen werden.
- Schließlich können zur Bestimmung der Gleichheit auch formale Verfahren wie bei der Abhängigkeits-Analyse (vgl. Abschnitt 4.4.2) verwendet werden, falls nur lineare Ausdrücke der Variablen vorkommen.

Vir gehen hier nicht näher auf die formalen Verfahren ein, da in Abschnitt 5.2.2
urch eine zusätzliche Bedingung eine einfache und exakte Alias-Analyse für be-
timmte Indexausdrücke in Schleifen möglich wird. Und für die anderen Ausdrücke
eicht meist eine der oben genannten einfachen Analysen aus.

5.1.4 Koprozessor-Synthese

Wir kommen nun zur eigentlichen Koprozessor-Synthese: Wenn ein Kandidat nur zulässige Anweisungen enthält und keine MAY-Aliase zwischen im Kandidaten definierten Variablen oder Feldelementen bestehen, können wir für ihn einen *azyklischen Datenfluß-Graphen* synthetisieren. Aus diesem wird dann ein konkreter Koprozessor für die rekonfigurierbare Hardware erzeugt, indem die Operatoren durch Module einer FPGA-Bibliothek instanziert werden. So ist die Synthese nicht auf einen bestimmten FPGA-Typ beschränkt, sondern für alle Baustein-Familien geeignet, für die eine Bibliothek mit Realisierungen der Operatoren vorliegt.

In praktischen Systemen setzt die Synthese auf der Zwischensprache eines Übersetzers auf. So können vor der Synthese die gebräuchlichen Übersetzer-Optimierungen (Erkennen gemeinsamer Teilausdrücke, Entfernen toten Codes, Konstantenfaltung etc.) angewendet werden.

Dann wird für jeden Basisblock des Kandidaten ein Datenfluß-Graph generiert, der die lokalen Datenabhängigkeiten repräsentiert. (Dieser entspricht der DAG-Darstellung der Befehle des Basisblocks.) Da keine Aliase vorkommen, können Feldzugriffe wie einfache Variablen behandelt werden. Die Befehle der Programmiersprache (bzw. der Zwischensprache) werden im Datenfluß-Graphen durch Hardware-Operatoren gleicher Funktionalität ersetzt. Einige Befehle, z. B. Konstanten oder Schiebe-Operatoren, benötigen eigentlich gar keine Hardware-Operatoren (und folglich keine FPGA-Logik-Blöcke), da sie einfach durch entsprechende Verdrahtung der Eingabe- bzw. Ausgabe-Signale realisiert werden können. Im Datenfluß-Graphen werden sie jedoch als *Pseudo-Operatoren* ohne Hardware-Bedarf beibehalten.

Da die einzigen im Kandidaten vorkommenden Kontrollkonstrukte bedingte Anweisungen sind, können wir durch Kontrollfluß-Datenfluß-Transformationen (vgl. Abschnitt 3.2.1) alle Kontrollabhängigkeiten eliminieren. Wir erhalten einen Datenfluß-Graphen für den gesamten Rumpf, in dem durch Multiplexer die richtigen Werte der bedingt zugewiesenen Variablen ausgewählt werden.

Dieser Graph wird schließlich durch regelbasierte Transformationen hardwarespezifisch optimiert. Die folgenden Regeln reduzieren entweder den Hardware-Bedarf oder ermöglichen eine schnellere Auswertung:[4]

- Reduziere die Baumhöhen arithmetischer Ausdrücke durch Ausnutzen der Kommutativität und Assoziativität der Operatoren. (Dies verringert den Hardware-Bedarf und später die Pipeline-Tiefe).
- Reduziere die Datenpfad-Breite auf tatsächlich benötigte Bitlängen (z. B. bei Unterbereichstypen, bei Ergebnissen von Schiebeoperatoren oder bei der Maskierung höherwertiger Bits).
- Verwende vorzeichenlose Operatoren, wenn alle Operanden vorzeichenlos sind.

[4] Die Regeln entsprechen den optimierenden Programm-Transformationen der High-Level-Synthese (Abschnitt 3.2.1).

- Optimiere Operatoren mit konstanten Operanden (z. B. Schiebeoperator statt Multiplikation mit oder Division durch 2^N).
- Ersetze mehrere Operatoren in alternativen Pfaden durch einen Operator (vgl. PISYN-System, Abschnitt 4.3.2).

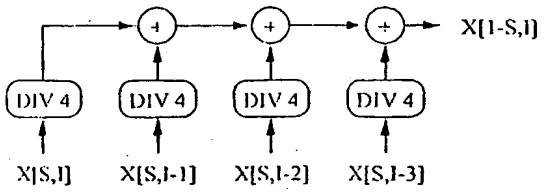


Abbildung 5.3: Azyklischer Datenfluß-Graph

Abbildung 5.3 zeigt den azyklischen Datenfluß-Graphen für das Beispielprogramm aus Abbildung 5.2. Er berechnet aus den Eingaben `X[S,I]`, `X[S,I-1]`, `X[S,I-2]` und `X[S,I-3]` die Ausgabe `X[1-S,I]`. (Konstante Operanden wurden in dieser Darstellung in die Operator-Knoten integriert.)

Durch Instanzieren der Operatoren eines Datenfluß-Graphen erhalten wir ein reines Schaltnetz, an dessen Ausgängen bei korrekt anliegenden, stabilen Eingabewerten nach einer festen Verzögerungszeit die Ausgaben des Kandidaten anliegen. Es ist offensichtlich, daß zum Einsatz dieser Koprozessoren weitere Steuerlogik nötig ist, die stabile Eingabewerte an die entsprechenden Ports des Koprozessors anlegt und nach der Verzögerungszeit die Ausgabeports ausliest. Wir werden die Anforderungen an diese Steuerlogik in Abschnitt 5.2.2 für die komplizierteren Pipeline-Schaltungen präzisieren.

Diese kombinatorischen Koprozessoren erreichen den größten innerhalb eines Kandidaten möglichen Parallelitätsgrad, da nur voneinander abhängende Operatoren serialisiert werden. Auch werden durch die direkte Wiederverwendung von Zwischenergebnissen im Vergleich zu einer Software-Ausführung Schreib- und Lesecoperationen eingespart. Dafür muß aber für jede arithmetische oder logische Operation des Kandidaten ein Hardware-Operator instanziert werden, der dann pro Koprozessor-Aufruf nur einmal benutzt wird. Deshalb reicht die Parallelität innerhalb eines Kandidaten kaum aus, um den Konfigurierungsaufwand für solche Koprozessoren zu rechtfertigen, wenn sie im Programm nicht sehr häufig ausgeführt werden.

Um dieses Problem zu entschärfen, wird im PRISM-System versucht, den Hardware Bedarf und die Verzögerung durch Logik-Minimierung zu reduzieren. Alternativ könnte durch eine günstige Ablaufplanung und Operator-Zuteilung Hardware wiederverwendet und eingespart werden (vgl. High-Level-Synthese, Abschnitt 3.2.1). Schließlich könnten auch äußere Schleifen *teilweise* ausgerollt werden, um größere Kandidaten zu erhalten, deren Hardware-Bedarf dann wiederum durch eine entsprechende Ablaufplanung und Wiederverwendung reduziert werden muß. Dies ist

aber schwierig, da die Datenabhängigkeiten bei teilweise ausgerollten Schleifen durch eine einfache Alias-Analyse nicht mehr erfaßt werden können. Eine einheitliche Behandlung der ganzen Schleife ist auf diese Weise nicht möglich.

Deshalb verzichten wir auf diese Optimierungen und analysieren im nächsten Abschnitt alle Abhängigkeiten einer Schleife, ohne sie auszurollen. So können wir ihre Struktur systematisch nutzen und *alle* Operatoren in einem Koprozessor in jeder Iteration wiederverwenden, was zu einer erheblichen Leistungssteigerung führt.

5.2 Pipelines für Schleifen ohne Abhängigkeiten

Die im vorhergehenden Abschnitt entwickelte Koprozessor-Synthese ist auf Kandidaten ohne Schleifen beschränkt. Rekonfigurierbare Hardware eignet sich jedoch besonders für reguläre, iterative Berechnungen, wie wir in Abschnitt 3.4.2 gesehen haben. Da diese aber in höheren Programmiersprachen durch Schleifen ausgedrückt werden, sind Koprozessoren für Schleifen besonders wichtig.

Außerdem sollte ein Koprozessor die Parallelität der Hardware möglichst gut ausnutzen, um eine Anwendung zu beschleunigen. Dies kann sehr gut durch Pipeline-Schaltungen, in denen alle Pipeline-Stufen gleichzeitig aktiv sind, erreicht werden. Wir kombinieren deshalb beide Anforderungen, indem wir Schleifen durch Pipelines ausführen: Verschiedene Schleifen-Iterationen werden von den Pipeline-Stufen überlappt bearbeitet. Dazu muß die Schleife jedoch vektorisierbar sein. Das heißt, es dürfen keine Abhängigkeiten zwischen den Iterationen bestehen, die ihre Überlappung in einer Pipeline verhindern.

Deshalb zeigen wir in diesem Abschnitt, welche Abhängigkeiten analysiert werden müssen und wie für vektorisierbare Schleifen effiziente Koprozessoren durch Pipelining synthetisiert werden können. Zunächst müssen die geeigneten Schleifen jedoch ausgewählt und normalisiert werden.

5.2.1 Schleifen-Auswahl und -Normalisierung

Wir betrachten nur innere Schleifen als Hardware-Kandidaten, da sie das größte Beschleunigungspotential enthalten. Äußere Schleifen würden kompliziertere Koprozessoren erfordern und sollten besser in Software bearbeitet werden.[5] Außerdem kommen nur FOR-Schleifen in Frage, da bei anderen Schleifentypen bereits durch die Abbruchbedingung eine Abhängigkeit von der vorhergehenden Iteration besteht. Dabei muß der Rumpf einer FOR-Schleife ein Hardware-Kandidat im Sinne von Abschnitt 5.1 sein.

Nach dieser Vorauswahl müssen die Schleifen und die in ihnen referenzierten Felder für die Vektorisierung und die Pipeline-Synthese normalisiert werden. Dabei wer-

[5] Dies schließt natürlich nicht aus, daß kleine innere Schleifen durch vollständiges Ausrollen (vgl. Abschnitt 5.1.2) vor der Auswahl der Kandidaten eliminiert werden können.

len spezielle Felder, die *Vektoren*, definiert und identifiziert. Nur diese können in ?ipelines verarbeitet werden.

Bereichs-Normalisierung

Zunächst werden das von der Schleifenvariable durchlaufene Intervall und die Indexbereiche der referenzierten Felder in einfacher zu analysierende Normalformen transformiert:

- Substituiere die Schleifenvariable so, daß die neue Variable in der ersten Iteration mit 0 initialisiert und in den folgenden Iterationen jeweils um 1 erhöht wird. Die Schleife hat dann die Form `FOR I:=0 TO N DO ... END`.
- Transformiere die Indexbereiche der Felder so, daß die untere Grenze aller Bereiche 0 ist.

In der Schleife L2 unseres Beispielprogrammes aus Abbildung 5.2 muß nur die Schleifenvariable I durch I+3 substituiert und das durchlaufene Intervall entsprechend angepaßt werden. Das Ergebnis ist in Abbildung 5.4 dargestellt.

```
...
S := 0;       (* X[0] is source array *)
FOR ITER := 1 TO MAXITER DO (* L1 *)
  FOR I := 0 TO N-3 DO      (* L2 *)
    X[1-S,I+3] := 0;
    X[1-S,I+3] := X[1-S,I+3] + X[S,I+3] DIV 4;
    X[1-S,I+3] := X[1-S,I+3] + X[S,I+2] DIV 4;
    X[1-S,I+3] := X[1-S,I+3] + X[S,I+1] DIV 4;
    X[1-S,I+3] := X[1-S,I+3] + X[S,I] DIV 4;
  END;
  S := 1 - S;
END
```

Abbildung 5.4: FIR-Filter nach Schleifenvariablen-Substitution

Felder und Vektoren

Für die nächste Transformation müssen wir zwischen verschiedenen Klassen von Feldern unterscheiden. Bei der einfachen Koprozessor-Synthese aus Abschnitt 5.1 konnten wir noch alle Felder einheitlich betrachten. Bei der Pipeline-Synthese nehmen jedoch Felder, auf die in Abhängigkeit von der Schleifenvariable zugegriffen wird, eine Sonderrolle ein. Wir müssen deshalb die folgenden Klassen unterschiedlich behandelter Daten präzise definieren:

Definition 1 (Skalare Daten) Skalare Daten *sind diejenigen Variablen, die von den üblichen Operatoren einer Programmiersprache als Einheit verarbeitet werden.*

Anmerkung: Diese skalaren Daten sind intern als Bitfelder repräsentiert, deren Länge meist der Wortlänge des verwendeten Prozessors entspricht. Durch entsprechende Typdefinitionen können auch kürzere Wortlängen (etwa 8 oder auch 1 Bit) definiert werden.

Definition 2 (Felder) *Ein* eindimensionales Feld *(Reihung, Array) ist die in allen imperativen Programmiersprachen bekannte Zusammenfassung mehrerer skalarer Daten desselben Typs. Ein Feld* `A` *der Länge N besteht aus den Elementen* `A[0]` *bis* `A[N-1]`, *die in aufeinanderfolgenden Adressen im Speicher abgelegt sind. Ein* zweidimensionales Feld `B` *ist ein Feld, dessen Elemente (Teilfelder)* `B[I]` *selbst eindimensionale Felder sind. Die skalaren Daten werden mit* `B[I][J]` *oder* `B[I,J]` *referenziert. Es ergibt sich also eine zeilenweise Speicherung. Wir bezeichnen* `J` *als ersten und* `I` *als zweiten Index von* `B`. *Analog werden* mehrdimensionale Felder *definiert. In allen Fällen werden konstante Feldlängen vorausgesetzt.*

Anmerkung 1: Diese Definition entspricht MODULA-Arrays mit normalisierten Indexbereichen, wie sie durch die oben vorgestellte Normalisierung erzeugt werden.

Anmerkung 2: Bei speziellen Befehlen, die einzelne Bits eines Wortes manipulieren (z. B. INCL auf BITSET), zählt nicht das Bit, sondern das ganze Wort als skalares Datum, da es in der Programmiersprache immer als Ganzes manipuliert wird. Eine Variable vom Typ BITSET wird also als skalares Datum, und *nicht* als Feld einzelner Bits betrachtet.

Definition 3 (I-Abhängigkeit in Schleife L) *Wir bezeichnen ein eindimensionales Feld als* I-abhängig in L, *wenn in L an mindestens einer Stelle mit einem von der Schleifenvariablen I abhängenden Indexausdruck auf das Feld zugegriffen wird. Mehrdimensionale Felder sind* in der n-ten Dimension I-abhängig, *falls dies für den Indexausdruck der n-ten Dimension gilt.*

Definition 4 (Vektor in Schleife L) *Ein eindimensionales I-abhängiges Feld ist ein* Vektor in L.

Definition 5 (Vektorfeld in Schleife L) *Ein Vektor kann auch ein eindimensionales Teilfeld eines mehrdimensionalen Feldes sein, das nur in der ersten Dimension I-abhängig ist. Wir sprechen dann von einem* Vektorfeld in L.

Definition 6 (Allgemeines Feld in Schleife L) *Ein in L nicht I-abhängiges Feld ist ein* allgemeines Feld in L.

lit diesen Definitionen wird für jede Schleife zwischen I-abhängigen und allgemei-
n Feldern unterschieden. Die Werte der I-abhängigen Felder, bei denen in jeder
hleifeniteration auf andere Elemente zugegriffen wird, werden bei der Vektorisie-
ng analysiert und in der Pipeline-Synthese berücksichtigt.

ies ist für die Zugriffe auf allgemeine Felder nicht nötig. Da ihre Indexausdrücke
cht von der Schleifenvariable abhängen und wegen möglicher Aliase sowieso nicht
n Zwischenergebnissen abhängen dürfen (vgl. Abschnitt 5.1.3), wird in jeder Ite-
tion dasselbe Element referenziert. Folglich könnten die Elemente der allgemeinen
lder in der Schleife durch skalare Variablen ersetzt werden. Vor der Schleife müßte
r Wert des Feldelementes an die skalare Variable zugewiesen werden, und nach der
hleife der Wert der skalaren Variable an das Feldelement. Da diese Transforma-
n immer möglich ist, können wir ohne Beschränkung der Allgemeinheit fordern,
ß in einer Schleife keine allgemeinen Felder vorkommen dürfen. Wir beschränken
se Forderung jedoch auf Zuweisungen an allgemeine Felder, da Elemente, die nur
lesen werden, wie skalare Eingaben behandelt werden können.

nzeptionelle Vektortransformation

gen der gerade geforderten Einschränkung für allgemeine Felder müssen wir
einem Kandidaten nur I-abhängige Felder betrachten. Wir wollen Hardware-
prozessoren entwerfen, die diese bearbeiten können. Um deren Steuerlogik
glichst einfach zu halten, beschränken wir uns auf die Adressierung von Vek-
en, also eindimensionalen I-abhängigen Feldern. Denn diese können direkt auf
eindimensionalen Adreßraum des lokalen Speichers der RE abgebildet werden.
der kommen aber auch mehrdimensionale I-abhängige Felder vor, die keine Vek-
felder sind, da sie nicht oder nicht nur in der ersten Dimension I-abhängig sind.
müssen in Vektoren oder Vektorfelder transformiert werden, um die Adressie-
g in einem Koprozessor zu vereinfachen. Dadurch wird auch gleichzeitig die zur
torisierung nötige Abhängigkeits-Analyse vereinfacht.

```
forall Felder F in L mit Dimension n > 1 do
  k := größte I-abhängige Dimension in F; /* k=0, falls I-unabhängig */
  if k > 1 then
    Transformiere k-dimensionales Teilfeld von F mit Feldlängen l_1, l_2,...,l_k
    in den Dimensionen 1 bis k in Vektor der Länge l_1 × l_2 × ... × l_k;
    /* Falls k=n, wird das ganze Feld transformiert. */
```

ildung 5.5: Algorithmus zur konzeptionellen Vektortransformation einer Schleife
it der Schleifenvariable I

Algorithmus in Abbildung 5.5 stellt eine solche Transformation dar: Er ver-
delt alle mehrdimensionalen Felder, die in einer höheren Dimension I-abhängig

sind, in Vektorfelder, so daß die I-Abhängigkeit nur noch in Vektoren (also in der ersten Dimension) auftritt. Dazu betrachten wir die I-abhängigen, mehrdimensionalen Teilfelder einfach als größere, unstrukturierte eindimensionale Felder der enthaltenen skalaren Daten. Dies ist eine rein *konzeptionelle Transformation*, die nur die Betrachtungsweise, nicht aber die tatsächliche Anordnung der Elemente im Speicher ändert. Sie nimmt die Adreßauflösung im Übersetzer (also die Abbildung auf den eindimensionalen Adreßraum) teilweise vorweg. Ist für ein n-dimensionales Feld die größte I-abhängige Dimension k, so wird das k-dimensionale Teilfeld als Vektor betrachtet. Wir erhalten ein (n-k+1)-dimensionales Vektorfeld, dessen eindimensionale Teilfelder Vektoren sind. Die Vektoren sind jeweils zusammenhängend in disjunkten Abschnitten gespeichert und können unabhängig voneinander bearbeitet werden.

Beispiel: Da in unserem durchgehenden Programmbeispiel keine konzeptionelle Vektortransformation notwendig ist, wollen wir sie am Beispiel des folgenden dreidimensionalen Feldes `X` erläutern:

```
X: ARRAY[0..L3-1],[0..L2-1],[0..L1-1] OF INTEGER
```

```
(1) X[J,  I,I+J]                X'[J,  I*L1+(I+J)] (normalisiert X'[J,I*(L1+1)+J])
(2) X[J+1,I,2]          ⇒       X'[J+1,I*L1+2]
(3) X[1,  2,I]                  X'[1,  2*L1+I]
(4) X[J,  I,I]                  X'[J,  I*L1+I] (normalisiert X'[J,I*(L1+1)])
```

Abbildung 5.6: Beispiel zur Vektor-Transformation

Abbildung 5.6 zeigt links Zugriffe auf `X` (mit Schleifenvariable I). Durch diese Zugriffe ist `X` in der ersten (1, 3, 4) und zweiten (1, 2, 4) Dimension I-abhängig, jedoch nicht in der dritten.[6] Folglich ist k=2, und das zweidimensionale Teilfeld mit den Größen L2 und L1 wird in einen eindimensionalen Vektor (der Größe L2*L1) transformiert. Wir erhalten das folgende neue Vektorfeld:

```
X': ARRAY[0..L3-1],[0..L2*L1-1] OF INTEGER
```

Die geänderten Feldzugriffe sind in Abbildung 5.6 rechts zu sehen. Lineare Ausdrücke in I treten jetzt nur noch in der ersten Dimension auf. □

Für die Vektorisierung müssen wir fordern, daß die Indexausdrücke in Zugriffen auf Vektor-Elemente nur linear von der Schleifenvariable abhängen. Um dies zu überprüfen, werden sie nach der konzeptionellen Vektortransformation nochmals wie in Abschnitt 5.1.3 normalisiert. Wir können die Ausdrücke dann in der Form $C + V + S \times I$ darstellen. Dabei bezeichnet C eine ganzzahlige Konstante, V einen Ausdruck von außerhalb der Schleife definierten Variablen, I die Schleifenvariable und $S \neq 0$ die ganzzahlige, konstante Schrittweite, wobei $C = 0$ oder $V = 0$ sein darf. Um einheitlich auf die Vektoren zugreifen zu können, fordern wir außerdem,

[6] Man beachte, daß die Dimension der Indizes *von rechts nach links* zunimmt.

ß V in allen Zugriffen auf einen Vektor (bzw. auf einen Teilvektor eines Vektor-des) gleich sein muß. So ist für einen Vektor der ganze Ausdruck $C + V$ in einer hleifenausführung konstant. Im Beispielprogramm aus Abbildung 5.4 ist dies der ll, da in allen Fällen $V = 0$ gilt.

lias-Analyse für Vektor-Elemente

un nutzen wir die oben getroffenen Einschränkungen, um eine einfache und exakte lias-Analyse für die Vektor-Elemente abzuleiten. Denn in Abschnitt 5.1.3 wurden ir für einfache Fälle Methoden zur Alias-Analyse von Feldzugriffen angegeben. Um ipeline-Schaltungen zu synthetisieren, ist aber für möglichst viele Fälle eine exakte lias-Analyse der Vektor-Elemente notwendig. Da wir forderten, daß V für alle ugriffe auf einen Vektor gleich sein muß, ist ein MAY-Alias durch andere Variablen s die Schleifenvariable bereits ausgeschlossen.

eispiel: In Abbildung 5.6 ist V in den normalisierten Ausdrücken 1 und 4 (mit hleifenvariable I) ungleich ($V = J$ und $V = 0$). Folglich ergibt sich für $J = 0$ ein lias, da der I-abhängige Index der ersten Dimension dann in beiden Ausdrücken eich ist. Wäre V in allen Ausdrücken gleich, könnte sich ein solcher MAY-Alias icht ergeben. □

ür die verbleibenden durch die Schleifenvariable verursachten Aliase kann nun eine akte Analyse durchgeführt werden. Sie überprüft für alle Zugriffe auf einen Vektor, b ihr Index zu einem Zeitpunkt denselben Wert annehmen kann, die Zugriffe also IAY-Aliase sind. Da alle Ausdrücke die Form $A_i = C_i + V + S_i \times I$ haben, muß für lle Paare (A_1, A_2) mit $C_1 \neq C_2$ oder $S_1 \neq S_2$ die folgende Gleichung gelöst werden.

$$A_1 = A_2 \tag{5.1}$$

$$\Leftrightarrow \quad C_1 + V + S_1 \times I = C_2 + V + S_2 \times I \tag{5.2}$$

s ergeben sich zwei Fälle, die einfach überprüft werden können:

- Für $S_1 = S_2$ folgt $C_1 = C_2$. Dies ist ein Widerspruch zur Annahme $C_1 \neq C_2$ oder $S_1 \neq S_2$. Also gibt es keine Lösung und auch keinen Alias.
- Für $S_1 \neq S_2$ folgt $I = \frac{C_2 - C_1}{S_1 - S_2}$. Falls I ganzzahlig ist und im Intervall [0..N] liegt, existiert ein MAY-Alias.

n unserem Beispielprogramm aus Abbildung 5.4 gilt in allen Fällen $S = 1$, so daß IAY-Aliase ausgeschlossen sind.

5.2.2 Vektorisierung

Nachdem eine FOR-Schleife normalisiert ist und auch überprüft wurde, daß sie nur ineare Vektorzugriffe ohne MAY-Aliase enthält, ist die eigentliche Vektorisierung

recht einfach. Wir müssen nur überprüfen, ob die Schleife in einer Pipeline ausgeführt werden kann. Dies geschieht durch eine Abhängigkeits-Analyse des Schleifenrumpfes. Eine explizite Transformation in Vektoranweisungen geschieht jedoch nicht. Vielmehr werden alle Anweisungen des Schleifenrumpfes implizit zu Vektoranweisungen, indem Pipeline-Operatoren für sie erzeugt werden (siehe Pipeline-Synthese, Abschnitt 5.2.3). Um die zu analysierenden Abhängigkeiten genau zu definieren, stellen wir zunächst das zugrundeliegende Pipeline-Ausführungsmodell vor.

Dabei erkennen wir, daß für die Vektorisierung eine schwächere Bedingung als für die Parallelisierung ausreicht, da die Iterationen in Pipelines zwar überlappt, aber immer noch in der ursprünglichen Reihenfolge ausgeführt werden, während bei Parallel- oder Vektorrechnern gar keine Aussage über die Ausführungsreihenfolge gemacht werden kann.

Pipeline-Ausführungsmodell

Die Pipeline-Schaltungen führen die Schleifenanweisungen auf der RE ohne Interaktion mit dem Wirt aus. Dafür sind (nach der Konfigurierung der RE) drei prinzipielle Phasen notwendig:

1. *LOAD:* Laden der Felder in den lokalen Speicher der RE und der skalaren Variablen in FPGA-Register

2. *EXEC:* Unabhängige Pipeline-Ausführung

3. *STORE:* Rückschreiben der Ergebnisse zum Wirt

Der gesamte Ablauf wird von einer *Pipeline-Steuerungseinheit* gesteuert, die im Abschnitt 6.3.1 vorgestellt wird. Das Verfahren könnte beschleunigt werden, indem die meist langsame Datenübertragung zwischen Wirt und RE und die Pipeline-Ausführung überlappt werden.[7]

In der EXEC-Phase durchlaufen die im externen Speicher abgelegten Vektordaten den aus dem Schleifenrumpf aufgebauten Datenfluß-Graphen als Datenströme. Dabei erfolgt das Lesen aller Eingabevektoren und das Schreiben aller Ausgabevektoren gleichzeitig und mit der eigentlichen Datenverarbeitung überlappt. Auch die Verarbeitung selbst ist normalerweise in mehrere überlappende Pipeline-Stufen unterteilt. Deshalb werden die Ausgabewerte einer Iteration im externen Speicher erst gespeichert, nachdem schon mit der Berechnung einiger nachfolgender Iterationen begonnen wurde. Folglich dürfen als Eingaben keine Werte verwendet werden, die in einer vorangehenden Iteration verändert worden sein könnten. Wir müssen also fordern, daß keine *echten schleifengetragenen Abhängigkeiten* auftreten.

[7] Diese Erweiterung wird in Abschnitt 7.4.1 diskutiert.

Analyse schleifengetragener Abhängigkeiten

Da äußere Schleifen sequentiell in Software ausgeführt werden, sind die von ihnen getragenen Abhängigkeiten irrelevant. Wir müssen also nur die von der zu vektorisierenden inneren Schleife getragenen Abhängigkeiten berücksichtigen. Wie oben erklärt wurde, sind von diesen im Pipeline-Ausführungsmodell wiederum nur die *echten* relevant, da die Reihenfolge der Schreib- und Leseoperationen erhalten bleibt. Andernfalls müßten auch die Anti- und Ausgabeabhängigkeiten betrachtet werden.

Zur Analyse benötigen wir zunächst die Mengen INPUT (Ausdrücke, die im Schleifenrumpf nur gelesen werden oder gelesen werden, bevor sie geschrieben werden) und OUTPUT (Ausdrücke, die im Schleifenrumpf geschrieben werden).[8] Da aus allen Anweisungen des Schleifenrumpfes ein Datenfluß-Graph erzeugt wird, können wir den ganzen Rumpf als eine große Anweisung mit der Eingabemenge INPUT und der Ausgabemenge OUTPUT betrachten. Abhängigkeiten entstehen dann, wenn ein Ausdruck in OUTPUT ein skalares Datum (also eine Speicheradresse) referenziert, das in einer späteren Iteration von einem Ausdruck in INPUT referenziert wird. Dies ist bei skalaren Variablen, die in OUTPUT und INPUT enthalten sind, immer der Fall: Da sie nur eine Speicheradresse referenzieren, wird ein Ausgabewert immer in der direkt folgenden Iteration weiterverwendet.

Vektoren und Vektorfelder müssen hingegen genauer analysiert werden. Jedoch ist nur der Index der ersten Dimension relevant, da nach der konzeptionellen Vektortransformation nur in dieser Dimension eine I-Abhängigkeit möglich ist. Für einen Vektor ergibt sich also genau dann eine echte Abhängigkeit, wenn die Indexfunktion $A_1(I)$ eines Zugriffs aus OUTPUT denselben Wert wie die Indexfunktion $A_2(J)$ eines Zugriffs (auf denselben Vektor) aus INPUT in einer späteren Iteration ($J > I$) annehmen kann. Dazu muß folgende Gleichung gelten:

$$A_1(I) = A_2(J) \tag{5.3}$$

$$\Leftrightarrow \quad C_1 + V + S_1 \times I = C_2 + V + S_2 \times J \tag{5.4}$$

$$\Leftrightarrow \quad S_1 \times I - S_2 \times J = C_2 - C_1 \tag{5.5}$$

Diese *lineare diophantische Gleichung* hat genau dann ganzzahlige Lösungen, wenn die folgende ggT-Bedingung gilt:

$$ggT(S_1, S_2) | (C_2 - C_1) \tag{5.6}$$

Außerdem müssen die Lösungen im richtigen Intervall liegen, also die folgende Zusatzbedingung erfüllen:

$$0 \leq I < J \leq N \tag{5.7}$$

Falls die leicht zu überprüfende ggT-Bedingung (5.6) nicht gilt, ist eine Abhängigkeit ausgeschlossen. Andernfalls unterscheiden wir wiederum zwei Fälle:

[8] Unter Ausdrücken verstehen wir hier skalare Variablen und normalisierte Vektorzugriffe

- Falls $S_1 = S_2$, definieren wir $S := S_1$. Dann gilt folgende Gleichung:

$$S \times (I - J) = C_2 - C_1 \tag{5.8}$$
$$\Leftrightarrow \quad S \times (J - I) = C_1 - C_2 \tag{5.9}$$
$$\Leftrightarrow \quad J - I = \frac{C_1 - C_2}{S} \tag{5.10}$$
$$\Leftrightarrow \quad J = I + \frac{C_1 - C_2}{S} \tag{5.11}$$

  Wir müssen also prüfen, ob es Paare $(I, J) = (I, I + \frac{C_1 - C_2}{S})$ gibt, die die Zusatzbedingung erfüllen. Dann liegt eine Abhängigkeit vor. Dies ist für $0 < \frac{C_1 - C_2}{S} \leq N$ der Fall, kann also ebenfalls einfach überprüft werden.

- Für $S_1 \neq S_2$ läßt sich die Zusatzbedingung nur mit dem aufwendigen Separabilitätstest bestimmen. Man könnte auch den einfacheren, aber ungenauen Banerjee-Test anwenden oder in allen Fällen eine Abhängigkeit annehmen.

Beispiele: In unserem Beispielprogramm (Abbildung 5.4) werden die Elemente von `X[S]` nur gelesen und das Element `X[1-S,I+3]` nur geschrieben. Es bestehen also gar keine schleifengetragenen Abhängigkeiten, so daß dieses Beispiel sowohl in unserem Sinne vektorisierbar als auch parallelisierbar (für einen Parallel- oder Vektorrechner) ist.

Der Unterschied zwischen den beiden Eigenschaften wird an der folgenden Schleife deutlich:

```
FOR I := 0 TO N DO
  X[I] := X[I] + X[I+1];
END
```

Sie ist nicht parallelisierbar, da die Zuweisung von sich selbst anti-abhängig ist. Das heißt, daß der in einer Iteration definierte Wert `X[I]` nicht in den Speicher geschrieben werden darf, *bevor* die Eingaben der im ursprünglichen Programm vorhergehenden Iteration gelesen wurden. Denn eine solche Vertauschung der Iterationen würde zu falschen Ergebnissen führen. Die Schleife ist jedoch vektorisierbar, da keine *echte* Abhängigkeit existiert. Der definierte Wert `X[I]` darf nämlich *beliebig spät* in den Speicher geschrieben werden, da er nicht weiterverwendet wird. Eine Überlappung der Lese-, Berechnungs- und Speicher-Phasen der Iterationen ist zulässig. □

5.2.3 Pipeline-Synthese

Bisher haben wir FOR-Schleifen normalisiert und analysiert, ob sie vektorisierbar sind. Dieser Abschnitt beschreibt nun, wie für vektorisierbare Schleifen Pipelineschaltungen synthetisiert werden. Dazu wird zuerst — wie in Abschnitt 5.1.4 — ein zyklischer Datenfluß-Graph des Schleifenrumpfes generiert, in den danach Eingaberegister und Pipeline-Stufen eingefügt werden.

Wir haben bereits bei der Diskussion der einfachen kombinatorischen Koprozessoren erwähnt, daß durch zusätzliche Logik stabile Eingabewerte für den Koprozessor erzeugt werden müssen. Dies ist am einfachsten durch getaktete Register möglich, deren Werte gezielt gesetzt werden können. Wir werden also *Eingaberegister* zu dem Datenfluß-Graphen hinzufügen. Dieser wird dann als Schaltung bezeichnet, da er mehr als nur kombinatorische Operatoren enthält.

Die skalaren Eingabevariablen werden — nach Abschnitt 5.2.2 — in der LOAD-Phase direkt in FPGA-Register geschrieben. Für jede skalare Variable in INPUT wird also ein Register, das nur einmal vor Beginn der EXEC-Phase initialisiert wird, zur Schaltung hinzugefügt. Die Vektordaten werden jedoch in den lokalen Speicher kopiert. Sie müssen also vor der Berechnung des Schleifenrumpfes gelesen und ebenfalls in FPGA-Registern gespeichert werden. Wir könnten nun einfach auch für alle Vektoreingaben unabhängige Eingaberegister allokieren. In der EXEC-Phase müßten dann aber vor jeder Schleifeniteration alle Werte nacheinander durch je einen *Vektor-Eingabe-Port* aus dem Speicher gelesen und in die Register geschrieben werden.

Dabei sollte aber berücksichtigt werden, daß bei Schleifen in jeder Iteration — also in jedem *Pipeline-Takt* — ein neuer Satz Vektoreingaben gelesen wird. Falls ein Vektor-Element bereits in einem früheren Takt gelesen wurde, müßte es nicht nochmals eingelesen werden. Es sollte vielmehr in einem FPGA-Register gespeichert und *wiederverwendet* werden, da ein Zugriff auf den externen Speicher viel langsamer als ein Registerzugriff ist. Dies ist möglich, da Vektoreingaben, die in verschiedenen Iterationen dieselben Vektor-Elemente referenzieren, folgendermaßen bestimmt werden können:

Definition 7 (Äquivalenz von Vektoreingaben) *Zwei Eingaben desselben Vektors mit den Zugriffsfunktionen* $A_1 = C_1 + V + S_1 \times I$ *und* $A_2 = C_2 + V + S_2 \times I$ *sind* äquivalent, *falls* $S_1 = S_2$ *und* $C_1 \bmod S = C_2 \bmod S$ *(mit* $S := S_1$*). Die* Distanz *der Eingaben ist dann* $D = \frac{C_1 - C_2}{S}$.

Nach dieser Definition liegen die Adressen (Indizes) aller durch Eingaben einer Äquivalenzklasse referenzierten Vektor-Elemente in derselben Restklasse *mod S*. Sie können durch *einen* Eingabe-Port und ein Schieberegister im FPGA, das die letzten Werte speichert, realisiert werden. Der Eingabe-Port liest die Elemente mit der Schrittweite S, und die Schieberegister erzeugen *verzögerte* Versionen dieses Eingabe-Datenstroms. Die Distanz D bestimmt die Anzahl der Register zwischen zwei Eingaben, also die Verzögerungsstufen.

Da vor der ersten Schleifeniteration nicht alle Vektoreingaben auf einmal gelesen werden, benötigen wir nun noch eine zusätzliche, mehrere Takte dauernde Ausführungsphase FILL, in der die Schieberegister gefüllt werden. In der EXEC-Phase müssen nun nur noch — überlappend mit den Berechnungen der vorhergehenden Iteration — die neuen Werte der Schieberegister eingelesen werden.

Abbildung 5.7 zeigt die Beispiel-Schaltung für unser normalisiertes Beispielprogramm aus Abbildung 5.4. Für die Zugriffe auf `X[S]`, die alle in einer Äquivalenz-

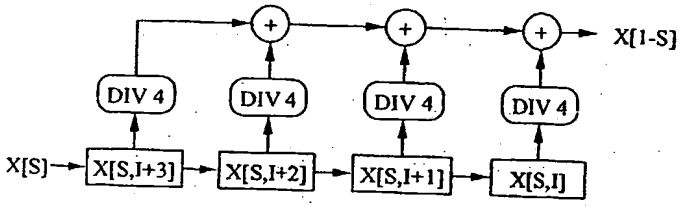


Abbildung 5.7: Beispiel-Schaltung mit Eingabe-Schieberegister

klasse liegen, wurde ein Eingabe-Schieberegister eingefügt. Es ist in der Abbildung schattiert dargestellt.

Trotz der Optimierung der Vektor-Eingaben ist eine so erzeugte Schaltung noch recht ineffizient, da der ganze Schleifenrumpf in einem Takt ausgewertet wird. Wegen der hohen Signalverzögerungen ist die erreichbare Taktfrequenz klein und der Durchsatz gering. Die Leistung einer Pipeline wird aber wesentlich durch ihren Durchsatz, also durch die Geschwindigkeit, mit der die Schleifeniterationen bearbeitet werden, bestimmt. Dagegen ist die Zeit zum Füllen der Pipeline, in der keine Ergebnisse geliefert werden, bei langen Schleifen vernachlässigbar.

Folglich muß der Durchsatz einer Schaltung durch Pipelining erhöht werden. Dies erhöht ihren Parallelitätsgrad, indem nicht nur Eingaben, Berechnungen und Ausgaben, sondern auch die ursprünglichen Schleifeniterationen selbst untereinander überlappt werden. Die damit gleichzeitig verbundene Erhöhung der Latenz der Pipeline ist unbedeutend.

Der Durchsatz kann allerdings nicht beliebig erhöht werden, da äußere Faktoren einen minimalen Wert für die Taktperiode T_C vorgeben. Wir müssen also zunächst diesen minimalen Wert für T_C bestimmen.

Bestimmung der Taktperiode T_C

T_C hängt von der Anzahl der Eingabe- und Ausgabe-Ports für Vektoren ab. Da wir von einer Architektur mit einer Speicherbank ausgehen (siehe Abschnitt 6.1), kann immer nur ein Schreib- oder Lesezugriff in einem Speicherzyklus stattfinden. Deshalb müssen diese Speicherzugriffe innerhalb eines Pipeline-Taktes nacheinander durchgeführt werden. Wir berechnen T_C also durch folgende Formel:

$$T_C := N_E \times T_E + N_A \times T_A \tag{5.12}$$

Dabei bezeichnen N_E und N_A die Anzahl der Eingabe- bzw. Ausgabe-Ports, und T_E und T_A die Zykluszeit zum Lesen bzw. Schreiben eines Wortes im lokalen Speicher. T_E und T_A hängen von der Geschwindigkeit des lokalen RAM und den Frequenzen der auf der RE verfügbaren Taktsignale ab. Gleichung (5.12) zeigt, daß die Anzahl der Vektor-Ports den Durchsatz der Pipeline bestimmt, und nicht die Zahl der

peratoren. Es sollten vor allem unnötige Ausgabe-Ports vermieden werden, da das chreiben eines Wortes in den RAM länger als das Lesen dauert.

ei der Bestimmung von T_C muß die Signalverzögerung innerhalb der Hardware-perator selbst nicht berücksichtigt werden, da wir *mehrstufige Operatoren* it ausreichend vielen internen Pipeline-Stufen aus einer entsprechenden Modul-ibliothek auswählen können.

ipelining

Vir fügen nun weitere Pipeline-Stufen in eine Schaltung ein, um den Durchsatz zu höhen. Dabei ist es nur sinnvoll, so viele Register hinzuzufügen, wie zum Erreichen er Taktperiode T_C nötig sind, da die Register Hardware-Ressourcen verbrauchen.

Vie in Abschnitt 4.3.4 erwähnt, muß beim Pipelining die Anzahl der eingefügten egister auf allen nicht-zyklischen Pfaden von einem Eingabe-Register zu einem perator und zu den Ausgängen gleich sein. Diese Registerzahl entspricht der *Latenz* es Operators, also der Verzögerung in Pipeline-Takten, mit der gültige Werte an den perator ankommen. Falls mehrstufige Operatoren in der Schaltung vorkommen, nüssen auch die internen Register beim Pipelining berücksichtigt werden. Dann muß lso auch ein Registerausgleich durchgeführt werden. (Zur Vereinfachung können wir nnehmen, daß sich bei allen mehrstufigen Operatoren ein Register direkt vor dem usgang befindet.)

Diese Forderungen können durch ein einfaches Verfahren zum Einfügen von Pipeline-egistern erfüllt werden. Dabei werden alle Knoten (d. h. die Operatoren und Ein-aberegister der Schaltung) topologisch sortiert durchlaufen, so daß jeder Knoten rst dann bearbeitet wird, wenn alle seine Eingaben bereits bearbeitet wurden. So ird für jeden Knoten die akkumulierte Signalverzögerung und die Latenz berechnet. alls die Signalverzögerung größer als T_C wird oder die Latenz der Eingaben ungleich st, werden Register in die entsprechenden Eingabekanten eingefügt. Allerdings wird abei die Zahl der Register nicht optimiert, was zu sehr großen Schaltungen mit vie-en unnötigen Registern führen kann. Wir kommen deshalb in Abschnitt 5.4.2 auf ieses Problem zurück.

Durch Pipelining ändert sich neben der Signalverzögerung auch das externe Zeit-erhalten einer Schaltung: Die Ausgabe-Werte einer Iteration liegen nicht mehr am Ende des Taktes, in dem die Eingaben bereitgestellt wurden, an den Ausgängen an, sondern erst einige Takte später. Folglich muß die Pipeline nach dem Einle-sen der letzten Eingabewerte weiter getaktet werden, bis sie leergelaufen ist, und die Ausgaben müssen verzögert gespeichert werden. Die Latenz muß also von der Pipeline-Steuerungseinheit beachtet werden.

Deshalb fügen wir eine Kette hintereinandergeschalteter Flipflops zur Schaltung inzu, deren Länge der maximalen Latenz eines Ausgangs entspricht. Das erste Flipflop wird vom VALID-Signal gesetzt, sobald alle Eingabe-Register gültige Werte haben (also am Beginn der EXEC-Phase) und zurückgesetzt, nachdem die Eingabe-

Daten aller Iterationen gelesen wurden. Damit ist jedes Flipflop ein *Gültigkeits-Bit*, das signalisiert, ob die Werte einer Pipeline-Stufe gültig sind oder nicht. Die EXEC-Phase wird erst beendet, wenn das letzte Gültigkeitsbit zurückgesetzt wird (Signal VALID_OUT). Dieses Signal steuert auch die Vektor-Ausgabe-Ports.

Abbildung 5.8 zeigt die Beispiel-Schaltung, nachdem zwei zusätzliche Pipeline-Stufen (zur Verringerung der Signalverzögerung der drei hintereinandergeschalteten Addierer) und die Gültigkeits-Bits eingefügt wurden.

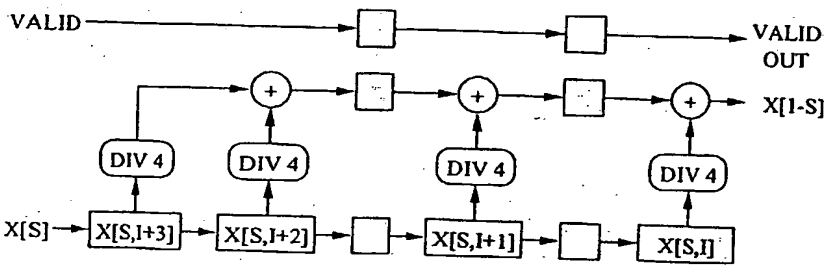


Abbildung 5.8: Beispiel-Schaltung nach Pipelining

5.3 Pipelines für Schleifen mit regulären Abhängigkeiten

Im vorhergehenden Abschnitt wurde ein neuartiges Verfahren zur Synthese von Pipeline-Schaltungen aus Schleifen eines sequentiellen Eingabeprogramms vorgestellt. Die Pipelines nutzen die Hardware-Operatoren durch eine überlappte Ausführung der Schleifen-Iterationen optimal aus.

Jedoch ist das Verfahren auf vektorisierbare Schleifen ohne echte schleifengetragene Abhängigkeiten beschränkt. Diese Klasse von Schleifen ist zwar größer als die der parallelisierbaren Schleifen, aber immer noch recht eingeschränkt, so daß auch nur bestimmte Pipelines synthetisiert werden können. Die in Pipelines einfach zu realisierenden Präfix-Operatoren (vgl. Abschnitt 4.3.2) können beispielsweise nicht synthetisiert werden. Deshalb erweitern wir in diesem Abschnitt die Klasse der vektorisierbaren Schleifen, indem auch bestimmte echte schleifengetragene Abhängigkeiten zugelassen werden. Die Pipeline-Synthese wird entsprechend erweitert, damit unter anderem auch die Präfix-Operatoren generiert werden können.

Schließlich wird kurz erläutert, wie mit dem erweiterten Synthese-Verfahren neben FOR-Schleifen auch WHILE-Schleifen bearbeitet werden können.

5.3.1 Erweiterte Vektorisierung

Wir haben in Abschnitt 5.2.2 festgestellt, daß in vektorisierbaren Schleifen keine Ausgabe einer vorangehenden Iteration als Eingabe der aktuellen Iteration verwendet werden darf. Diese Bedingung wird nun abgeschwächt. Denn wir können trotz einer solchen Abhängigkeit eine Pipeline für die Schleife synthetisieren, wenn die weiterzuverwendenden Ausgabewerte direkt in Pipeline-Registern gespeichert und weiterverwendet werden können. Sie dürfen also nicht erst (mit unbekannter Verzögerung) in den externen Speicher geschrieben und von dort wieder eingelesen werden. Diese direkte Weiterverwendung ist jedoch nur dann möglich, wenn die Abhängigkeit (nach der Initialisierung) in jeder Iteration und mit der gleichen Distanz auftritt. Wir nennen sie dann *regulär* Abhängigkeit und können sie durch ein Schieberegister, dessen Länge der Distanz der Abhängigkeit entspricht, in der Pipeline realisieren. Die Klasse der vektorisierbaren Schleifen wird also auf solche mit regulären schleifengetragenen Abhängigkeiten erweitert.

Echte Abhängigkeiten, die von skalaren Variablen verursacht werden, sind immer regulär mit der Distanz $D = 1$, da die Werte direkt in der nächsten Schleifeniteration weiterverwendet werden. Sie können durch ein einzelnes Register in einer Pipeline realisiert werden.

Bei Vektor-Abhängigkeiten sind diejenigen Fälle der Analyse aus Abschnitt 5.2.2 regulär, für die $S := S_1 = S_2$ und die Zusatzbedingung $0 < \frac{c_1 - c_2}{S} \leq N$ gilt. Dann liegt nämlich für alle Paare $(I, J) = (I, I + \frac{c_1 - c_2}{S})$ im Intervall [0..N] eine Abhängigkeit mit der festen Distanz $D = \frac{c_1 - c_2}{S}$ vor.

Abhängigkeiten mit $S_1 \neq S_2$ sind jedoch *irregulär* und können nicht behandelt werden.

Beispiel: Nach einer kleinen Änderung enthält die am Ende von Abschnitt 5.2.2 vorgestellte Beispielschleife eine echte schleifengetragene Abhängigkeit:

```
FOR I := 0 TO N DO
  X[I+2] := X[I] + X[I+1];
END
```

Nun ist jede Iteration von den zwei vorhergehenden Iterationen echt abhängig. Die Abhängigkeiten sind aber regulär (mit den Distanzen $D = 1$ und $D = 2$), da für alle Indexausdrücke $S = 1$ gilt. Deshalb können wir im nächsten Abschnitt für diese Schleife eine Pipeline erzeugen. □

5.3.2 Pipeline-Synthese mit Rückkopplungszyklen

Nachdem reguläre Abhängigkeiten definiert und analysiert wurden, wird nun die in Abschnitt 5.2.3 vorgestellte Pipeline-Synthese auf Schleifen mit solchen Abhängigkeiten erweitert. Dazu ist nach dem Einfügen der Eingaberegister und vor dem Pipelining ein weiterer Bearbeitungsschritt nötig. Er fügt Rückkopplungs-Zyklen

für die regulären Abhängigkeiten in die Schaltung ein. Durch diese Rückkopplungen ergeben sich auch für das nachfolgende Pipelining neue Einschränkungen, die in manchen Fällen die Erzeugung effizienter Pipelines verhindern können.

Wir betrachten zuerst den einfacheren Fall der Abhängigkeiten, die von skalaren Variablen verursacht werden. Danach stellen wir eine Transformation vor, die die komplizierteren Vektor-Abhängigkeiten auf den einfacheren Fall zurückführen, und definieren eine zusätzliche Pipelining-Bedingung.

```
TYPE SCARD = [0..65535]; (* 16-bit cardinal *)
VAR  S:      [0..1];
     X:      ARRAY[0..1],[0..N] OF SCARD;
     RAND:   SCARD;
...
S    := 0; (* X[0] is source array *)
RAND := 1; (* initialize random number *)
FOR ITER := 1 TO MAXITER DO (* L1 *)
  FOR I := 0 TO N-3 DO      (* N(L2) *)
    IF (15 IN BITSET(RAND)) THEN
      RAND := SCARD(BITSET(2 * RAND) / {14,13,12,10,8,6,3,0});
    ELSE
      RAND := 2 * RAND;
    END;
    X[1-S,I+3] := 0;
    X[1-S,I+3] := X[1-S,I+3] + X[S,I+3] DIV 4;
    X[1-S,I+3] := X[1-S,I+3] + X[S,I+2] DIV 4;
    X[1-S,I+3] := X[1-S,I+3] + X[S,I+1] DIV 4;
    X[1-S,I+3] := X[1-S,I+3] + X[S,I] DIV 4;
    X[1-S,I+3] := X[1-S,I+3] + RAND;
  END;
  S := 1 - S;
END
```

Abbildung 5.9: Erweitertes normalisiertes Programmbeispiel FIR-Filter

Zunächst erweitern wir aber unser durchgehendes Programmbeispiel aus Abbildung 5.4. In Abbildung 5.9 wurde das normalisierte Programm durch ein linear rückgekoppeltes Schieberegister zur Generierung einer Zufallszahl ergänzt.[9] Die skalare Variable RAND verursacht eine reguläre schleifengetragene Abhängigkeit. Abbildung 5.10 zeigt die Schaltung für dieses Beispiel nach dem Einfügen der Eingabe-Schieberegister. Man sieht, wie die bedingte Zuweisung an RAND durch einen von der Bedingung gesteuerten Multiplexer realisiert wird.

[9] In MODULA-2 bezeichnet / den XOR-Operator und {14,13,12,10,8,6,3,0} den Bitvektor '0111010101001001' für den Datentyp BITSET.

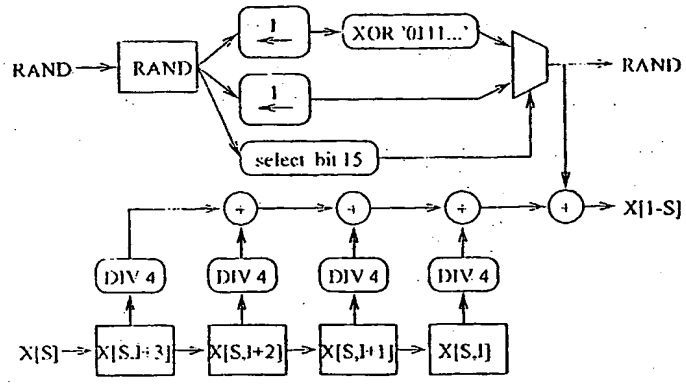


Abbildung 5.10: Schaltung für erweitertes FIR-Filter

Skalare Rückkopplungsvariablen

Wir bezeichnen skalare Variablen, die eine schleifengetragene Abhängigkeit verursachen, als *skalare Rückkopplungsvariablen*, da für sie *Rückkopplungen* in die Pipeline-Schaltung eingefügt werden müssen. Dies soll anhand der Abbildung 5.10 erläutert werden: Da in die Schaltung noch keine zusätzlichen Pipeline-Stufen eingefügt wurden, wird nach dem Pipeline-Ausführungsmodell (nach dem Laden der Schieberegister) in jedem Pipeline-Takt eine Schleifeniteration berechnet. Dazu werden normalerweise die Werte, mit denen die skalaren Eingaberegister vor der EXEC-Phase initialisiert wurden, verwendet. Handelt es sich aber um eine Rückkopplungsvariable, gilt die Initialisierung nur für die erste Iteration, also den ersten Pipeline-Takt. Ab dem zweiten Takt muß das Register immer den im vorhergehenden Takt berechneten Wert enthalten. Wir erreichen dies, indem ein Multiplexer vor dem Register eingefügt wird. Er wählt gesteuert von einem externen Steuersignal — während der LOAD-Phase den Eingabewert und während der EXEC-Phase den rückgekoppelten Ausgabewert. Dies wurde in Abbildung 5.11 für die Variable RAND durchgeführt.

Durch diese Rückkopplungen entsteht eine zyklische Schaltung. Ihre stark zusammenhängenden Komponenten nennen wir *Rückkopplungszyklen*. Dabei können durch Variablen mit gegenseitigen Abhängigkeiten auch Zyklen mit mehreren Registern entstehen. Das Schaltwerk bleibt dabei synchron, wenn die Taktfrequenz die Signalverzögerung zwischen den Registern berücksichtigt.

Die erzeugte Schaltung funktioniert jedoch noch nicht korrekt, da jedes skalare Eingabe-Register seinen Wert nur bei seiner Initialisierung und — falls es ein Rückkopplungsregister ist — in der EXEC-Phase ändern darf, nicht aber während der Initialisierung der anderen Eingänge und während der FILL-Phase. Dies muß durch eine korrekte Ansteuerung der Clock-Enable-Eingänge der Register gewährleistet werden. Die dafür nötige Logik wird im übernächsten Abschnitt beim Pipelining

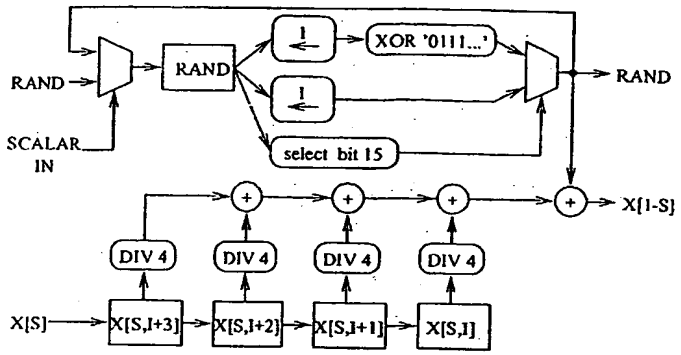


Abbildung 5.11: Schaltung für erweitertes FIR-Filter mit Rückkopplung

hinzugefügt.

Vektor-Rückkopplungen

Zuvor betrachten wir die Abhängigkeiten durch Vektoren: Statt für sie direkt Rückkopplungen zu erzeugen, transformieren wir vor der Pipeline-Synthese alle Abhängigkeiten durch Vektoren in solche durch skalare Hilfsvariablen. Auf diese Weise werden Vektor-Eingabeports gespart, und der Durchsatz der Pipelines wird nach Gleichung (5.12) erhöht. Denn in Schleifen mit regulären Vektor-Abhängigkeiten (d. h. in linearen Rekurrenzen) verwenden nur wenige Iterationen am Anfang der Berechnung Vektor-Eingabewerte, während alle folgenden Iterationen die rückgekoppelten Werte verwenden. Eingabeports für diese Vektoren würden also kaum genutzt werden.

Die Transformation wird auf dem Zwischencode des Programms durchgeführt, aber hier zur Vereinfachung auf der Quellsprach-Ebene erläutert: Im Schleifenrumpf wird jede Vektor-Eingabe, die von einer Ausgabe abhängt, durch eine neue Hilfsvariable ersetzt. Dann werden an das Ende des Schleifenrumpfes für alle Abhängigkeiten mit Distanz $D = 1$ neue Zuweisungen angehängt, die den Hilfsvariablen die Werte zuweisen, von denen die ursprünglichen Vektor-Eingaben abhängen. Für Abhängigkeiten mit einer Distanz $D > 1$ werden zusätzliche Hilfsvariablen erzeugt und am Ende des Schleifenrumpfes zugewiesen, so daß die indirekten Abhängigkeiten durch eine entsprechende Zahl von Hilfsvariablen ersetzt werden. Schließlich werden vor der Schleife Zuweisungen eingefügt, die den Hilfsvariablen die ursprünglichen Vektor-Eingabewerte zuweisen. So kann die oben für skalare Rückkopplungsvariablen entwickelte Methode für alle Abhängigkeiten eingesetzt werden.

Beispiel: Der linke Teil der Abbildung 5.12 zeigt die Beispiel-Schleife aus Abschnitt 5.3.1. Mit den angegebenen Initialisierungen berechnet sie die Fibonacci-Zahlen und schreibt sie in einen Vektor. Die Schleife wird folgendermaßen in den in Abbildung

```
X[0] := 0;
X[1] := 1;
FOR I := 0 TO N DO
   X[I+2] := X[I] + X[I+1];
END
```

$\Rightarrow$

```
X[0] := 0;
X[1] := 1;
TMP1 := X[0];
TMP2 := X[1];
FOR I := 0 TO N DO
   X[I+2] := TMP1 + TMP2;
   TMP1   := TMP2;
   TMP2   := X[I+2];
END
```

Abbildung 5.12: Transformation der Fibonacci-Schleife

5.12 rechts gezeigten Programmabschnitt transformiert: Die Vektor-Eingaben `X[I]` und `X[I+1]` werden durch die skalaren Hilfsvariablen `TMP1` und `TMP2` ersetzt. Da `TMP2` (`X[I+1]`) direkt, d. h. mit Distanz $D = 1$, von der Vektor-Ausgabe `X[I+2]` abhängt, wird am Schleifenende `TMP2 := X[I+2]` gesetzt. `TMP1` (`X[I]`) hängt mit Distanz $D = 2$ von `X[I+2]` ab. Deshalb wird `TMP1` der Wert von `X[I+2]` in der vorhergehenden Iteration, also der aktuelle Wert von `TMP2` zugewiesen. Dabei müssen die Zuweisungen in der richtigen Reihenfolge erfolgen.

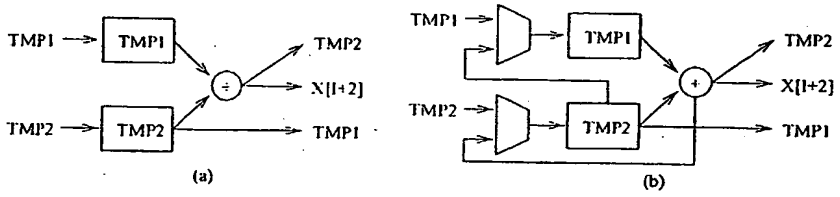


Abbildung 5.13: Schaltungen für Fibonacci-Schleife

Abbildung 5.13 zeigt die resultierende azyklische Schaltung (a) und die Schaltung mit Rückkopplungen (b). Durch die Hilfsvariablen `TMP1` und `TMP2` entsteht ein zweistufiger Rückkopplungszyklus. Die Pipeline-Ausführung wird also ermöglicht, da die berechneten Werte nicht nur im lokalen Speicher (über den Ausgang `X[I+2]`), sondern auch zur direkten Wiederverwendung in den Registern `TMP1` und `TMP2` gespeichert werden. Da in dieser Schaltung jedoch nur ein Operator vorkommt, lohnt sie sich nur als Bestandteil einer größeren Schaltung. □

Pipelining

Für das Pipelining der Schaltungen ergibt sich durch die Rückkopplungszyklen eine weitere Einschränkung: Wie wir in Abschnitt 4.3.4 gesehen haben, dürfen innerhalb der Zyklen einer Schaltung keine Register eingefügt werden. Wir können also die Signalverzögerung (und damit die erreichbare Taktperiode $T_\cdot$) nicht mehr durch zusätzliche Pipeline-Stufen beliebig verringern. Innerhalb der Zyklen kann die

Verzögerung nur durch Retiming verringert werden.[10]

Folglich kann die durch Formel (5.12) berechnete Taktperiode nicht erreicht werden, wenn in Rückkopplungszyklen höhere Signalverzögerungen auftreten. Wir berechnen diese Verzögerung, indem wir für alle kombinatorischen Pfade in dem Zyklus die erwarteten Signalverzögerungen der Operatoren aus einer Modul-Bibliothek für die verwendete FPGA-Familie entnehmen und aufsummieren. Der in Formel (5.12) berechnete Wert für T_C wird dann folgendermaßen korrigiert:

$$T_C := max(T_C, \text{max. Signalverzögerung der Rückkopplungszyklen}) \quad (5.13)$$

In den azyklischen Teil der Schaltung können nun wie in Abschnitt 5.2.3 soviele Stufen wie nötig eingefügt werden, um die Taktperiode T_C zu erreichen.

Wie bereits weiter oben erwähnt wurde, muß beim Pipelining noch Clock-Enable-Logik für die skalaren Eingaberegister generiert werden. Dies ist für die nicht-rückgekoppelten Register recht einfach: Da sie nur einmal initialisiert werden, verwenden sie ein externes Eingabe-Steuersignal als Clock-Enable-Signal. Wir brauchen zwar für jedes Register ein eigenes Steuersignal, können dafür aber einen gemeinsamen Eingabebus für alle Register verwenden, da das Steuersignal nur das gewünschte Register aktiviert.

Die rückgekoppelten Register müssen jedoch bei der Initialisierung und in der EXEC-Phase aktiviert werden. Bei mehrstufigen Pipelines darf sich der gespeicherte Wert aber auch nur dann ändern, wenn die Werte der Pipeline-Stufe, in welcher der Rückkopplungszyklus liegt, gültig sind. Andernfalls könnte der Eingabe- oder der letzte Ausgabewert zu früh überschrieben werden. Deshalb wird das Clock-Enable-Signal dieser Register durch eine disjunktive Verknüpfung des jeweiligen Eingabe-Steuersignals und des Gültigkeitsbits der entsprechenden Stufe (Latenz) aktiviert. Im Gegensatz dazu sind die normalen Pipeline-Register immer aktiviert, da sie in jedem Takt einen neuen Datensatz verarbeiten und zwischenspeichern.

Abbildung 5.14 zeigt die Schaltung für das erweiterte FIR-Filter, nachdem eine zusätzliche Pipeline-Stufe und die Clock-Enable-Logik eingefügt wurden. Das Register `RAND` hat die Latenz 1 und wird deshalb nach der Initialisierung erst wieder aktiviert, wenn die erste Pipeline-Stufe im unteren Teil der Schaltung gefüllt ist, damit am letzten Addierer die Daten einer Iteration gleichzeitig anliegen.

Wir haben gesehen, daß die Bedingung für die Rückkopplungszyklen die erreichbare Leistung der Pipelines einschränkt. Man könnte deshalb versuchen, das Programm auf höherer Ebene zu transformieren, um den Durchsatz trotz Rückkopplungszyklen zu erhöhen. Da aber keine Ansätze, die solche Transformationen automatisch durchführen, bekannt sind, wurden sie in dieser Arbeit nicht weiter verfolgt.

Alternativ könnten wir eine synchrone Schaltung *skalieren*, also alle Register einer Pipeline — auch innerhalb von Rückkopplungszyklen — durch k Register ersetzen. Dann läßt sich durch Retiming auch die Verzögerung in den Zyklen auf etwa

[10] Dann muß allerdings auch die Initialisierung der verschobenen Register angepaßt werden, was jedoch nicht immer möglich ist.

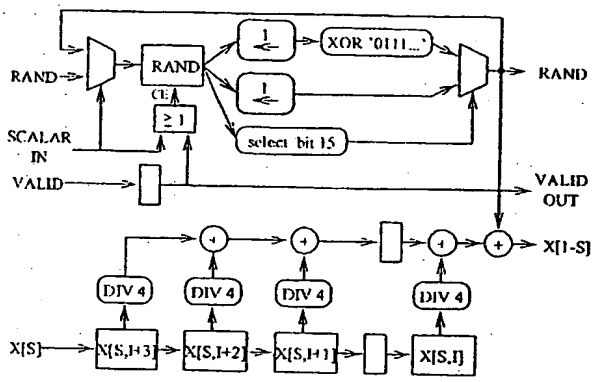


Abbildung 5.14: Schaltung für erweitertes FIR-Filter nach Pipelining

$1/k$ reduzieren. Jedoch dürfen dann auch nur alle k Takte Daten in die Pipeline gegeben werden, damit die Berechnung korrekt bleibt. Der Durchsatz bleibt also insgesamt gleich, aber es wird die k-fache Registerzahl gebraucht. Dieses Verfahren lohnt sich nur, falls k unabhängige Datenströme überlappt in die Pipeline gelesen werden können. Da bei unseren Pipelines meist die E/A-Bandbreite und nicht die Rückkopplungszyklen den Durchsatz der Schaltung begrenzen, lohnt sich der Aufwand für das Erkennen und Bereitstellen unabhängiger Datenströme jedoch nicht. Auch dieser Ansatz wurde deshalb hier nicht weiter verfolgt.

Eine weitere Möglichkeit zur Optimierung von Schleifen mit vielen bedingten Anweisungen stellt die in Abschnitt 4.3.3 vorgestellte spekulative Ausführung dar. Dabei wird der wahrscheinlichste Pfad eines Rückkopplungszyklus optimiert. Die Steuerung der Pipeline muß dann jedoch so erweitert werden, daß auch die anderen, langsameren Pfade ausgeführt werden können.

5.3.3 Behandlung von WHILE-Schleifen

Bis jetzt wurde nur die Pipeline-Synthese für normalisierte FOR-Schleifen betrachtet. Bei ihnen ist die Zahl der Schleifeniterationen vor Beginn der Ausführung bekannt und wird von der Pipeline-Steuerungseinheit überwacht. In der Pipeline muß also keine Hardware zur Evaluierung der Abbruchbedingung generiert werden. Dasselbe gilt natürlich für WHILE-Schleifen, die in normalisierte FOR-Schleifen transformiert werden können. Dies ist der Fall, wenn eine ganzzahlige Schleifenvariable in jeder Iteration um einen konstanten Wert erhöht oder erniedrigt wird und die Schleife abbricht, wenn diese Variable einen außerhalb der Schleife festgelegten Grenzwert über- oder unterschreitet.

Mit den vorgestellten Techniken können wir aber keine WHILE-Schleife, deren Abbruchbedingung von in der Schleife berechneten Werten abhängt, behandeln. Dies

ist aber möglich, wenn wir für die Abbruchbedingung wie für eine gewöhnliche Anweisung Hardware erzeugen. Ist die Bedingung erfüllt, muß dies der Pipeline-Steuerungseinheit signalisiert werden, damit die Ausführung der Pipeline beendet wird. Außerdem müssen die Gültigkeitsbits verändert werden, damit die Werte der nachfolgenden, bereits begonnenen Iterationen nicht gespeichert werden. Bereits durchgeführte Änderungen der Werte in Rückkopplungsregistern können jedoch nicht mehr rückgängig gemacht werden. Deshalb dürfen lebendige skalare Rückkopplungsvariablen, deren Werte nach dem Schleifenabbruch also benötigt werden, in diesen Schleifen nur in der ersten Pipeline-Stufe stehen. Die Möglichkeiten, die Schaltung durch Pipelining zu optimieren, sind in diesem Fall also eingeschränkt.

Noch weniger Optimierungsmöglichkeiten gibt es in ganz allgemeinen WHILE-Schleifen, in denen auch keine Schleifenvariable, die in jeder Iteration um eine Konstante erhöht oder erniedrigt wird, vorkommt. In diesen Schleifen gibt es offensichtlich (per definitionem) keine Vektoren oder Vektorfelder. Da nach Abschnitt 5.2.1 keine Zuweisungen an allgemeine Felder erlaubt sind, dürfen im Schleifenrumpf folglich nur Zuweisungen an skalare Variablen auftreten. Falls diese in einer Schleife sinnvoll verwendet werden, führen sie jedoch zu Rückkopplungsschleifen. (Dies ist etwa bei der oben vorgestellten Fibonacci-Schleife der Fall.) Die Pipelining-Möglichkeiten sind also auch eingeschränkt.

Prinzipiell können also auch für WHILE-Schleifen Pipeline-Schaltungen synthetisiert werden. Da sich aber nur bei der Verarbeitung von Vektoren ein hoher Parallelitätsgrad durch Pipelining erreichen läßt, eignen sich FOR-Schleifen weitaus besser als Hardware-Kandidaten.

5.4 Registeroptimierung für FPGA-Pipelines

Bisher haben wir nur die Optimierung des Pipeline-Durchsatzes betrachtet. Bei der Diskussion des Pipelining in Abschnitt 5.2.3 wurde jedoch erwähnt, daß das einfache Pipelining-Verfahren viele unnötige Register einfügen kann. Da diese aber den Flächenverbrauch der Pipelines wesentlich mitbestimmen, ist eine Minimierung der Registerzahl bei beschränkten FPGA-Ressourcen sehr wichtig. Wir stellen deshalb in diesem Abschnitt eine neue Methode zur globalen Registeroptimierung vor. Zuvor wird kurz auf eine zusätzliche Möglichkeit, in Rückkopplungszyklen Register einzusparen, eingegangen.

5.4.1 Registerzusammenfassung

In vielen Fällen befindet sich ein Pipeline-Register direkt am Ausgang eines Rückkopplungszyklus. Dann werden in allen relevanten Takten, in denen die Eingabedaten gültig sind, in dieses und das Eingabe-Register dieselben Werte geschrieben, da der Multiplexer den rückgekoppelten Wert für das Eingaberegister wählt und sein Clock-Enable-Eingang durch das Gültigkeitsbit gesetzt ist. Deshalb können die bei-

den Register auch zusammengefaßt werden. In unserem durchgehenden Beispiel ist dies der Fall, wenn wir für das Register RAND die Latenz 0 festlegen und ein zusätzliches Register vor dem Ausgang RAND einfügen. Dieses kann dann mit dem Rückkopplungsregister RAND zusammengefaßt werden. Abbildung 5.15 zeigt das Ergebnis. Es ist funktional äquivalent zu Abbildung 5.14 und benötigt gleich viel Hardware. Effektiv wurde aber hinter dem Rückkopplungszyklus eine weitere Pipeline-Stufe eingefügt. Deshalb muß die Signalverzögerung des Zyklus am Ausgang RAND nicht mehr berücksichtigt werden.

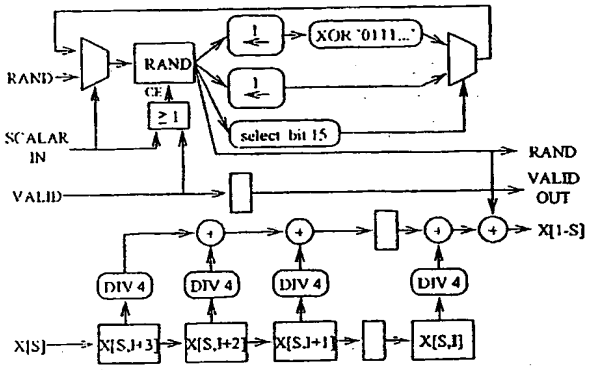


Abbildung 5.15: Beispiel-Schaltung nach Registerzusammenfassung

Da ein Register am Ende eines Rückkopplungszyklus also die Signalverzögerung verringert und auch die im nächsten Abschnitt vorgestellte Optimierung vereinfacht, führen wir die Zusammenfassung für alle Rückkopplungsregister durch. Wir müssen jedoch beachten, daß die Ausgabewerte jetzt intern um einen Takt verzögert werden. Beim Pipelining behandeln wir die Rückkopplungszyklen deshalb wie gewöhnliche Operatoren mit einem internen Register.

5.4.2 Lineares Programm zur optimalen Register-Verteilung

Mit dem gerade vorgestellten Verfahren können lokal einige Register eingespart werden. Ein weitaus größerer Effekt wird aber erreicht, wenn die Verteilung aller Register global optimiert wird. Deshalb entwickeln wir in diesem Abschnitt eine neue formale, auf ganzzahliger linearer Programmierung (engl. integer linear programming, ILP) basierende Optimierungsmethode zur Kombination von Register-Ausgleich und Pipelining für FPGA-Schaltungen. Sie löst das folgende

Register-Verteilungs-Problem:
Finde für eine gegebene Schaltung eine korrekt ausgeglichene Pipeline-

Schaltung, die mit der minimalen Anzahl FPGA-Flipflops die gegebene Taktperiode C_T erreicht.

Da in den gegebenen Schaltungen interne Register in den Rückkopplungszyklen initialisiert und direkt ausgelesen werden, dürfen sie bei der Optimierung nicht verändert werden. Folglich kann das Retiming-Verfahren (Abschnitt 4.3.4) nicht unverändert für das Verteilungs-Problem eingesetzt werden, da es nur das externe Verhalten der Schaltung gleich läßt, aber nicht die internen Register. Außerdem muß die Pipeline auch ausgeglichen werden, falls Operatoren mit internen Registern vorkommen. Aber auch das reine Ausgleichs-Verfahren von Gao (Abschnitt 4.3.4) ist nicht einsetzbar, da es für azyklische Programm-Graphen in Datenfluß-Rechnern entwickelt wurde und deshalb kein Pipelining durchführt und keine variablen Bitbreiten oder individuelle Verzögerungen der Operatoren berücksichtigt. Wir kombinieren deshalb Retiming und das Verfahren von Gao und erweitern die Methode so, daß sie auch die unregelmäßige Flipflop-Verteilung in vielen FPGA-Familien berücksichtigt.

Vorverarbeitung und Notation

Beim Lösen eines ILP werden ganzzahlige Variablenwerte bestimmt, die eine lineare *Kostenfunktion* unter Berücksichtigung einer Menge linearer *Bedingungen* (Ungleichungen) minimiert. Ist ein kombinatorisches Optimierungsproblem als ein solches ILP formuliert, kann es mit den Simplex- und Branch-and-Bound-Algorithmen effizient gelöst werden.

Wir wollen nun das Register-Verteilungs-Problem für eine Schaltung mit Rückkopplungen, für welche die Registerzusammenfassung aus Abschnitt 5.4.1 durchgeführt wurde, als ILP formulieren. Dazu betrachten wir die Schaltung formal als Graphen $G = (N, E)$ mit der Knotenmenge N und der Kantenmenge $E \subset N \times N$. Desweiteren ist $I \subset N$ die Menge der Eingaberegister, $O \subset N$ die Menge der Ausgabeknoten, $AI \subset I$ die Menge der Vektor-Eingaberegister, AI_0 ein spezielles, ausgewähltes Vektor-Eingaberegister, und $P \subset N$ die Menge der Pseudo-Operatoren, die keine Logik enthalten (z. B. konstante Schiebeoperatoren).

Um eine Kostenfunktion und Bedingungen aufstellen zu können, muß G vorverarbeitet werden: Zuerst wird jeder Rückkopplungszyklus durch einen *Superknoten* ersetzt, da die Zyklen nicht durch zusätzliche Register verändert werden dürfen. Wir erhalten einen azyklischen Graphen. Die *Knotenlatenz* NL_i, also die Zahl der internen Pipeline-Stufen, wird bei den Superknoten (wegen der Registerzusammenfassung) auf 1 gesetzt. Für die rein kombinatorischen Operatoren sei $NL_i = 0$, und für die mehrstufigen Operatoren entspreche NL_i der Anzahl der internen Register.

Danach werden die Schieberegister der verzögerten Vektoreingaben entfernt, da sie Gegenstand der Optimierung sind. Stattdessen wird eine Kante von dem ersten Eingaberegister zu dem Operator, an dem die verzögerte Eingabe verwendet wurde, eingefügt. Ihre *Kantenlatenz* $EL_{i,j}$ wird auf die benötigte Anzahl der Verzögerungsstufen (die Distanz der Vektorzugriffe) gesetzt. Für alle anderen Kanten ist $EL_{i,j} = 0$.

Wir entfernen auch die Eingänge selbst, da sie durch ihre Register repräsentiert werden, und die für die Optimierung irrelevanten Konstanten. N enthält dann also nur noch Operatoren, Superknoten, Eingaberegister und Ausgabeknoten. Abbildung 5.16 zeigt G für unser Beispiel. Dabei gilt $NL = 0$ und $EL = 0$, falls nichts anderes vermerkt ist. Der Superknoten für den Rückkopplungszyklus um RAND ist doppelt eingerahmt.

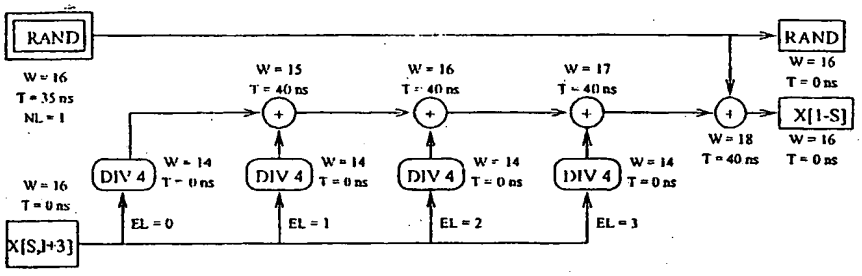


Abbildung 5.16: Vorverarbeiteter Graph G

Für die Optimierung sind noch weitere Eigenschaften der Schaltung relevant: Die Wortlänge der Operator-Ausgänge W_i wird verwendet, um die genaue Anzahl der gebrauchten Flipflops zu bestimmen, und $T_{i,j}$ ist die Signalverzögerung eines Operators j bezüglich seiner Eingabe von Knoten i. Für Knoten j, die Rückkopplungszyklen oder mehrstufige Operatoren repräsentieren, ist $T_{i,j}$ die Verzögerung vom Eingang über die Kante (i, j) zum ersten internen Register.

Für die zu erwartende Verzögerung durch die Verdrahtung des FPGA wird eine konstante durchschnittliche Verzögerungen zu jedem $T_{i,j}$ addiert.[11] Um eine funktionsfähige Schaltung zu garantieren, muß $T_{i,j} \leq T_C$ für alle Kanten (i, j) gelten. In Abbildung 5.16 sind auch Beispielwerte für diese Eingabedaten eingetragen.[12]

Aus den Eingabedaten werden die folgenden Werte, die alle nicht-negative Ganzzahlen sind, berechnet: d_i ist die maximale akkumulierte Verzögerung zwischen einem Register und dem Ausgang von Operator i. r_i und s_i zählen die benötigten Register und werden in der Kostenfunktion verwendet. Schließlich ist l_i die Anzahl der Register auf einem Pfad von einem Vektor-Eingabeport zu Knoten i, d. h. seine Latenz. Weil l_i die Anzahl der Register, die auf allen Pfaden zum Knoten i eingefügt werden, festlegt, ist eine ausgeglichene Pipeline garantiert. Tabelle 5.1 faßt die verwendete Notation zusammen.

[11] Die tatsächlichen Verzögerungen durch Verdrahtung können nur sehr schwer genauer geschätzt werden.

[12] Es wurde die vereinfachende Annahme gemacht, daß die Verzögerungen für alle Eingangskanten eines Knotens gleich sind. Deshalb ist nur ein Wert T für jeden Knoten eingetragen.

Eingabewerte	
T_C	Taktperiode (in ns)
NL_i	Knotenlatenz des Knoten i
$EL_{i,j}$	Kantenlatenz der Kante (i,j)
W_i	Wortlänge des Ausgangs des Knoten i
$T_{i,j}$	Signalverzögerung (in ns) von Ausgang Knoten i zu Ausgang Knoten j
Berechnete Werte	
d_i	maximale Verzögerung zwischen Register und Ausgang des Knoten i (in ns)
r_i	Anzahl am Ausgang von Knoten i eingefügter Register
s_i	Anzahl durch Kombination mit Logik von Operator i eingesparter Register
l_i	Latenz bezüglich Vektor-Eingängen (in Takten)

Tabelle 5.1: Notation

Kostenfunktion

Nun wird das eigentliche ILP vorgestellt: Wir wollen die Zahl der eingefügten Flipflops minimieren, die durch die folgende Kostenfunktion berechnet wird:

$$C = \sum_{i \in N} r_i \cdot W_i - \sum_{i \in N \setminus I \setminus O \setminus P, NL_i = 0} s_i \cdot W_i \qquad (5.14)$$

Der erste Term berechnet die Flipflop-Zahl aller Knoten, indem die Produkte von r_i (Anzahl der Register, die in ausgehenden Kanten des Knoten i gebraucht werden) und W_i (Wortlänge des Operators) für alle Knoten i aufsummiert werden. Da Register in verschiedenen ausgehenden Kanten eines Knotens zusammengefaßt werden können, muß nur die Länge der längsten ausgehenden Register-Kette berücksichtigt werden. Dies ist die Zahl r_i. Die zweite Summe berechnet die Zahl der Flipflops, die keine zusätzlichen Logikblöcke benötigen, da sie mit der kombinatorischen Logik des Operators zusammen in einem Block realisiert werden können. Dies ist maximal für ein Register in einer Kette möglich ($0 \leq s_i \leq 1$), aber nicht bei Pseudooperatoren ohne kombinatorische Logik und bei mehrstufigen Operatoren. Diese Einsparung ist auch nur bei FPGA-Familien möglich, die kombinatorische Logik und Flipflops in einem Logikblock vereinen. Für andere Familien kann die zweite Summe einfach weggelassen werden.

Bedingungen

Die folgenden Bedingungen definieren die zulässigen Lösungen:
Für alle Eingaberegister i ist die akkumulierte Verzögerung 0:

$$\forall i \in I : d_i = 0 \qquad (5.15)$$

Für das ausgewählte Vektor-Eingaberegister AI_0, das als Bezugspunkt für die La-

tenzen der anderen Register verwendet wird, ist die Latenz 0:

$$l_{Al_0} = 0 \tag{5.16}$$

Es kann höchstens ein Register durch Kombination mit Operator-Logik eingespart werden. Und dies nur dann, wenn überhaupt ein Register instanziert wurde:

$$\forall i \in N : s_i \leq 1, s_i \leq r_i \tag{5.17}$$

Bei keinem Knoten darf die akkumulierte Verzögerung größer als die Taktperiode sein:

$$\forall i \in N : d_i \leq T_C \tag{5.18}$$

Die Kanten von G ordnen die Operator-Ausführung: Folglich muß für alle Kanten (i,j) die Latenz des Knoten j mindestens so groß wie die des Knotens i plus die interne Knotenlatenz von i sein:

$$\forall (i,j) \in E : l_j \geq l_i + NL_i \tag{5.19}$$

Die Anzahl der Register an einem Knoten-Ausgang wird durch seine eigene Latenz und die seines Nachfolgers bestimmt. Also ist r_i für alle Kanten (i,j) mindestens so groß wie die Differenz der Latenzen plus die Kantenlatenz minus der internen Latenz des Knotens i:

$$\forall (i,j) \in E : r_i \geq l_j - l_i + EL_{i,j} - NL_i \tag{5.20}$$

Die akkumulierte Verzögerung eines Knotens ist mindestens so groß wie das Maximum der Signalverzögerung von allen seinen Eingängen:

$$\forall (i,j) \in E : d_j \geq T_{i,j} \tag{5.21}$$

Für Kanten (i,j), in die kein Register eingefügt wird $(l_i = l_j)$, ist die akkumulierte Verzögerung des Knoten j mindestens die Summe der akkumulierten Verzögerung des Knoten i und der Signalverzögerung $T_{i,j}$. Für Kanten mit Registern ergibt sich keine weitere Einschränkung für d_j:[13]

$$\forall (i,j) \in E : d_j \geq T_{i,j} + d_i + T_C \cdot (l_i - l_j) \tag{5.22}$$

Registerplazierung

Die Kostenfunktion und die Bedingungen definieren eine optimale Lösung des ILP. Ist sie berechnet, wird sie folgendermaßen verwendet, um die Register tatsächlich in die Schaltung einzufügen:

[13] Für Kanten mit Registern ist $l_i - l_j < 0$. Daraus folgt $T_C \cdot (l_i - l_j) \leq -T_C$. Da $d_i \leq T_C$ immer gilt, ist $d_i + T_C \cdot (l_i - l_j) \leq 0$. Die Bedingung entspricht in diesem Fall also der Bedingung 5.21.

Zuerst werden r_i Register an den Ausgang jedes Knotens i angehängt. Dann werden alle Kanten (i,j) durch Kanten vom n-ten Register zum Knoten j ersetzt, wobei $n = l_j - l_i + EL_{i,j} - NL_i$ ist. Dadurch werden die Register in allen ausgehenden Kanten eines Knotens automatisch kombiniert. Die in Abschnitt 5.2.3 benötigte Latenz für die Register in Rückkopplungsschleifen entspricht der Latenz des zugehörigen Superknotens. Dieser Wert gilt — ebenso wie die berechnete Latenz der Vektor-Ports — immer relativ zu dem speziellen Vektor-Eingaberegister AI_0.

Abbildung 5.17 zeigt G mit den berechneten Werten für l, r und s (falls sie von 0 verschieden sind) und mit den eingefügten Registern. Wie in Abbildung 5.14 wurde eine Pipeline-Stufe zwischen die Addierer eingefügt. Man sieht, daß alle Operatoren hinter den eingefügten Registern und die Ausgaben die Latenz $l = 1$ haben. Außerdem kann nur das Register zwischen den beiden Addierern mit den Logik-Blöcken des linken Addierers kombiniert werden ($s = 1$). Das Ergebnis ist fast mit Abbildung 5.14 identisch. Jedoch wurde ein Register hinter den rechten DIV-Operator verschoben. Durch die Berücksichtigung der Wortlänge der Operatoren wurde erkannt, daß so zwei Flipflops eingespart werden.[14]

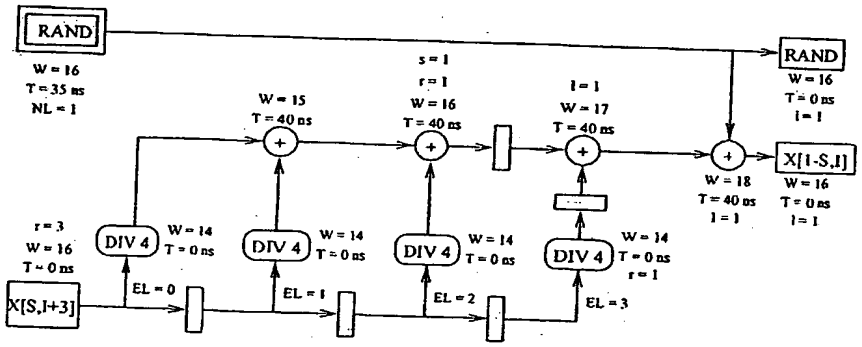


Abbildung 5.17: Graph G mit berechneten Werten und eingefügten Registern

Zeitkomplexität des Verfahrens

Da das ILP keine besonderen Eigenschaften erfüllt, muß im ungünstigsten Fall mit einer exponentiellen Laufzeit gerechnet werden. Dies ist jedoch vertretbar, da die Größe der verarbeitbaren Pipelines sowieso beschränkt ist und für ILP Lösungsverfahren existieren, die im allgemeinen effizient sind. Das Verfahren könnte jedoch auch so verändert werden, daß ein polynomielles Lösungsverfahren existiert: Auf die FPGA-spezifische Optimierung durch Kombination mit Operator-Logik (Bedingung 5.17) müßte allerdings verzichtet werden. Und die Bedingungen 5.20 bis 5.22 müßten — ähnlich wie beim allgemeinen Retiming — so umgeformt werden, daß nur noch

[14] Die Werte wurden mit dem "mixed IP-solver" [Ber92] berechnet.

Bedingungen der Form $x_i - x_j \leq b$ auftreten. Das resultierende ILP kann immer effizient gelöst werden, erfordert aber mehr Vorverarbeitungen der Schaltung.

5.5 Gesamtstruktur der Pipeline-Synthese

```
Wähle in P Kandidaten nach Algorithmus Abb. 5.19 aus; /* setzt KAND(L) */
Normalisiere Schleifen in P nach Algorithmus Abb. 5.20; /* ergibt N(L) */
forall Schleifen N(L) in P mit KAND(L) do /* normalisierte Kandidaten */
   Führe Alias-Analyse für N(L) durch [5.1.3, 5.2.1];
   if MAY-Aliase vorhanden then
      KAND(L) := falsch; /* Kandidat scheidet aus */
   Führe Abhängigkeits-Analyse für N(L) durch [5.2.2, 5.3.1];
   if irreguläre Abhängigkeit vorhanden then
      KAND(L) := falsch;
   Transformiere Vektor-Rückkopplungen in skalare Rückkopplungen [5.3.2];
   Synthetisiere Datenfluß-Graphen G für Rumpf von N(L) [5.1.4];
   Füge Eingaberegister in G ein [5.2.3];
   Füge Rückkopplungen in G ein [5.3.2];
   Fasse Rückkopplungsregister in G zusammen [5.4.1];
   Bestimme Taktperiode für G nach den Formeln (5.12) und (5.13);
   Bestimme optimale Registerverteilung für G [5.4.2];
   Füge Register, Gültigkeitsbits und Clock-Enable-Logik in G ein [5.2.3, 5.3.2];
   Gebe Schaltung G aus;
```

Abbildung 5.18: Algorithmus zur Pipeline-Synthese in Programm P

Nachdem alle notwendigen Methoden zur automatischen Synthese effizienter Pipeline-Schaltungen entwickelt wurden, kombinieren wir sie nun, um sie in einem Übersetzer einzusetzen. Abbildung 5.18 zeigt den resultierenden Algorithmus, wobei die eckigen Klammern die Abschnitte enthalten, in denen die entsprechenden Schritte vorgestellt wurden. Die Kandidaten-Auswahl und -Normalisierung werden in eigenen Abbildungen dargestellt, da sie hier ausführlicher als in den vorhergehenden Abschnitten beschrieben werden.

Abbildung 5.19 zeigt den Auswahlalgorithmus. Er setzt das Attribut KAND für alle Schleifen des Programms, die prinzipiell als Kandidaten in Frage kommen, also nur synthetisierbare Operationen und keine rekursiven Funktionsaufrufe oder nicht vollständig ausrollbare Schleifen enthalten (vgl. Abschnitt 5.1.2). Dazu wird das Hilfsattribut KOMP für alle Funktionen und Schleifen verwendet, die *Teil* eines Kandidaten sein können. Eine Schleife, die selbst Kandidat ist, kann auch Teil eines größeren Kandidaten sein, wenn sie vollständig ausrollbar ist. In diesem Fall hängt es letztlich von der Größe der Schleife und der verfügbaren Hardware ab, für

```
Konstruiere Aufrufgraphen A aller Funktionen F in P;
Erkenne rekursive Funktionen (stark zusammenhängende Komponenten) in A;
Konstruiere reduzierten Aufrufbaum A' durch Zusammenfassen dieser Kompo-
nenten;
forall F in A do
   if F rekursiv then
      KOMP(F) := falsch;
   else
      KOMP(F) := wahr;
forall F in A do (beginnend mit den Blättern in A' bis zur Wurzel)
   forall Schleifen L in F do (von innen nach außen)
      KAND(L) := wahr;
      if L enthält geschachtelte Schleife L' mit ¬ KOMP(L') or
      L ruft Funktion F' auf mit ¬ KOMP(F') or
      L enthält nicht-synthetisierbare Operation O then
         KAND(L) := falsch;
      if L vollständig ausrollbar then
         KOMP(L) := KAND(L);
      else
         KOMP(L) := falsch; /* Komponente muß ausrollbar sein */
      if ¬ KOMP(L) then
         KOMP(F) := falsch;
   forall Operationen O in F (ohne Schleifen) do
      if O ist Aufruf von F' mit ¬ KOMP(F') or
      O ist nicht-synthetisierbare Operation then
         KOMP(F) := falsch;
```

Abbildung 5.19: Algorithmus zur Kandidaten-Auswahl in Programm P

welchen Kandidaten die Pipeline-Synthese möglich und lohnend ist. Der Algorithmus berechnet KOMP für alle Funktionen des Aufrufbaums (von den Blättern zur Wurzel) sowie KAND und KOMP für alle Schleifen jeder Funktion von innen nach außen.

Nach der Auswahl werden die Schleifen durch den Algorithmus in Abbildung 5.20 normalisiert. Dabei werden die inneren Schleifen ausgerollt und die Funktionsaufrufe expandiert. Die ursprüngliche Programmversion jedes Kandidaten L wird als Software-Version SW(L) beibehalten, da später zur Laufzeit dynamisch zwischen Software- und Hardware-Implementierung ausgewählt wird. Die Hardware-Versionen N(L) aller Schleifen werden dann in normalisierte FOR-Schleifen oder normalisierte WHILE-Schleifen (mit Schleifenvariable) transformiert, soweit dies möglich ist. Schließlich werden die Normalisierungen (Abschnitt 5.2.1) durchgeführt.

```
Transformiere Indexbereiche aller Felder in Form [0..N];
Ersetze Feld-Zuweisungen durch einzelne Element-Zuweisungen;
Konstruiere A und A' wie in Algorithmus Abb. 5.19;
forall Funktionen F in A do (beginnend mit den Blättern in A' bis zur Wurzel)
  forall Schleifen L in F mit KAND(L) do (von innen nach außen)
    Kopiere L; Nenne ursprüngliche Version SW(L), neue N(L);
    Expandiere rekursiv alle Funktionsaufrufe in N(L);
    /* Verwende dabei für Schleifen L' in den Funktionen immer N(L'). */
    Rolle alle in N(L) vorkommenden inneren Schleifen vollständig aus;
    if N(L) enthält Schleifenvariable I then
      Substituiere alle zusätzlichen Schleifenvariablen;
      Normalisiere Schleifen-Initialisierung in die Form I:=0;
      Normalisiere auf Schrittweite 1;
      if Schleifenbedingung hat die Form I<=N then
        Transformiere N(L) in die Form /* FOR-Schleife */
         FOR I:=0 TO N DO ... END;
      else
        Transformiere N(L) in die Form /* normalisierte WHILE-Schleife */
         I:=0; WHILE <Schleifenbedingung> DO ...; I:=I+1 END;
      Führe konzeptionelle Vektortransformation in N(L) durch;
      /* siehe Abschnitt 5.2.1 */
      Normalisiere Indexausdrücke (möglichst in linearen Ausdruck von I);
    else
      Transformiere N(L) in die Form /* allgemeine WHILE-Schleife */
       WHILE <Schleifenbedingung> DO ... END;
```

Abbildung 5.20: Algorithmus zur Schleifen-Normalisierung in Programm P

.6 Bewertung

i diesem Kapitel wurde ein neuartiges Verfahren zur Synthese von Pipeline-haltungen aus Schleifen sequentieller Programme vorgestellt, das bekannte Vektosierungsmethoden erweitert und sie mit der High-Level-Synthese für rekonfigurierre Hardware kombiniert. Die Verfahren wurden erstmals so aneinander angepaßt, ıß sie in einen Übersetzer integriert werden können, um aus den Schleifen des Einbeprogramms automatisch Bitströme zur Konfigurierung der FPGAs zu erzeugen. as Verfahren beruht zunächst auf der Synthese einfacher kombinatorischer *Kozessoren* für den Schleifenrumpf. Ähnliche Verfahren wurden bereits im PRISM-stem (Abschnitt 4.1) und in einfachen High-Level-Synthese-Systemen verwendet. doch ist die Behandlung von Feldern sowie die damit zusammenhängende Aliasnalyse für höhere Programmiersprachen in unserem Verfahren neu.

Die einfachen Koprozessoren werden überlappt in einer Pipeline ausgeführt. Dazu müssen die geeigneten Schleifen *vektorisiert* werden. Durch die genaue Analyse der Feldzugriffe können so effiziente Schaltungen, die die Hardware-Operatoren parallel nutzen, synthetisiert werden. Dabei können auch für Schleifen, bei denen die in Abschnitt 4.3.3 vorgestellten Verfahren eine Abhängigkeit unterstellen müßten, Pipelines synthetisiert werden. Außerdem ermöglicht die Definition äquivalenter Eingaben und deren Kombination zu Eingabe-Schieberegistern die direkte Wiederverwendung von Eingabewerten, wodurch die Zahl der Vektor-Eingabeports und damit der Durchsatz der Pipeline signifikant beschleunigt werden kann. Weil die Reihenfolge der Schleifeniterationen in Pipelines erhalten bleibt, können auch Schleifen vektorisiert werden, die nicht parallelisiert werden könnten. Und da schließlich *reguläre schleifengetragene Abhängigkeiten* durch Rückkopplungen in der Pipeline realisiert werden, können wir sogar Schleifen mit diesen Abhängigkeiten behandeln. Insgesamt kann also eine große Klasse von Schleifen bearbeitet werden. Die dabei generierten Pipelines für rekonfigurierbare Hardware sind auch allgemeiner als Pipelines für Vektorrechner, da sie beliebige Operatoren — auch mit mehrstufigen Rückkopplungen — enthalten können, die zu beliebig tiefen Pipelines verkettet werden können. Allerdings ist die mögliche Komplexität der Operatoren in einem FPGA wesentlich kleiner als die der festen Operatoren einer Vektor-Einheit.

Da das genaue Zeitverhalten eines Koprozessors nicht vorgegeben ist, sondern nur die Funktionalität, können soviele Pipeline-Stufen eingefügt werden, wie zum Erreichen des durch die Ein-/Ausgabe-Bandbreite bestimmten Durchsatzes notwendig ist. Dabei sollten aber möglichst wenige Register eingefügt werden, um die knappen FPGA-Ressourcen effizient zu nutzen. Deshalb wurde ein neues formales Verfahren zur globalen Registeroptimierung entwickelt, das spezielle Eigenschaften bestimmter FPGA-Familien verwendet und nicht nur die Register, sondern die genaue Zahl der einzelnen FPGA-Flipflops minimiert.

Im Gegensatz zu Pipeline-Schaltungen nutzen die auf Standard-High-Level-Synthese beruhenden Koprozessor-Entwurfsverfahren ohne Vektorisierung die Parallelität der Hardware meist schlecht aus. Sie ist auf Operationen innerhalb eines Basisblocks beschränkt. Nur in Sonderfällen (ausrollbare Schleifen, Datenfluß-Kontrollfluß-Transformationen) sind auch blockübergreifende Optimierungen möglich. Da die Parallelität der Hardware in SP-Rechnern aber sehr gut genutzt werden muß, um eine Anwendung zu beschleunigen, ist unsere Pipeline-Synthese diesen Verfahren überlegen.

Trotzdem ist sie in verschiedener Hinsicht einfacher als High-Level-Synthese-Systeme: Da nur ungeschachtelte Schleifen verarbeitet werden, muß kein beliebiges Steuerwerk synthetisiert werden. Die Beschränkung auf eine Schleife ist jedoch — wie bereits erwähnt — für SP-Rechner kein Nachteil, da die weniger kritischen äußeren Schleifen in Software behandelt werden können. Außerdem ist die Bereitstellung und Zuordnung von Ressourcen einfach, da in der Pipeline für jeden Befehl ein Operator allokiert wird ("direkte Übersetzung"). Bis auf die Optimierung der alternativen Pfade wird auch keine Mehrfachnutzung (etwa durch die Verwendung von

ALUs) angestrebt. Auch dies ist für die Prozessorsynthese nachteilig, aber nicht für Pipelines, die von den kontinuierlichen Datenströmen optimal ausgenutzt werden. Schließlich ermöglicht die einheitliche Behandlung der Vektordaten, daß die Schaltungsteile zum Datenzugriff für alle Programme gleich sind und deshalb in einer generischen Pipeline-Steuerungseinheit manuell optimiert werden können.

Kapitel 6

Kombinierte Strukturprogrammierung

Nachdem wir im vorhergehenden Kapitel ein Verfahren zur Synthese von Pipeline-Schaltungen aus Schleifen eines sequentiellen Programms entwickelt haben, stellen wir nun das Gesamtsystem zur Übersetzung einer SP-Rechner-Anwendung vor. Es löst die in Abschnitt 2.2 genannten Teilprobleme Hardware/Software-Integration und -Partitionierung. Wir beginnen mit der Definition der relevanten Maschineneigenschaften und leiten die Struktur eines kombinierten Übersetzers für beide Teile des SP-Rechners ab. Danach diskutieren wir die Integration, also die Ansteuerung der Koprozessoren durch den Wirt über die Hardware/Software-Schnittstelle und die Pipeline-Steuerungseinheit auf der RE. Schließlich wird ein Partitionierungs-Verfahren entwickelt, das die Bedingungen, unter denen sich die Auslagerung der vektorisierbaren Schleifen auf die RE überhaupt lohnt, ermittelt und auswertet.
Die Hauptideen dieses Kapitels wurden bereits in [Wei96a] und [Wei96c] veröffentlicht.

6.1 Relevante Maschineneigenschaften

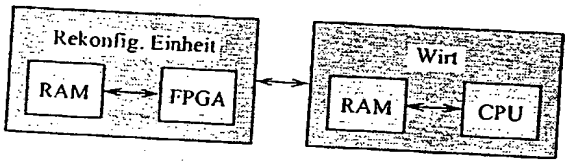


Abbildung 6.1: Allgemeines SP-Rechner-Modell

Um die Struktur eines Übersetzers möglichst konkret, aber dennoch unabhängig von einem bestimmten Rechner angeben zu können, definieren wir durch folgende vereinfachende Annahmen ein allgemeines SP-Rechner-Modell: Der Wirt ist ein beliebiger

in-Prozessor-Rechner, und die RE besteht aus *einem* FPGA und *einer* lokalen peicherbank (siehe Abbildung 6.1). Der Wirt kann über die Wirt-RE-Schnittstelle hreibend und lesend auf den lokalen Speicher der RE und direkt auf (ausgewählte) egister des FPGA zugreifen und das FPGA steuern. Ob der Zugriff auf den loka- n Speicher direkt oder nur über das FPGA erfolgen kann, ist irrelevant. Um die atenübertragung und -verarbeitung zu vereinfachen, gehen wir davon aus, daß die ortlänge des lokalen Speichers der RE und die Wortlänge des Hauptprozessors eich sind und der maximalen Größe der skalaren Daten nach Definition 1 (Seite) entsprechen. Außerdem kann das FPGA nur komplett in einem Schritt rekonfi- riert werden, wobei auch der lokale Speicher gelöscht wird.

er in den nächsten Abschnitten vorgestellte Übersetzer beruht nur auf diesen Ei- nschaften, ist aber unabhängig davon, wie sie realisiert werden. Da nur Para- eter für die Geschwindigkeit des Hauptprozessors, des FPGA und der Wirt-RE- hnittstelle verwendet werden, ist die genaue Architektur des SP-Rechners und die rwendete FPGA-Familie irrelevant.

die von uns verwendete Experimentierplattform diesem Modell entspricht, nnen die Pipeline-Synthese des vorhergehenden Kapitels und die Verfahren die- Kapitels in einem Übersetzer-Prototyp auf dieser Plattform getestet werden ehe Kapitel 7). Für größere SP-Rechner mit mehreren FPGAs und mehreren eicherbänken sind jedoch einige Erweiterungen des Modells notwendig. Diese wer- n in Abschnitt 9.4 erörtert.

2 Kombination der Übersetzungsmethoden

automatisch vollständige Anwendungen für einen allgemeinen SP-Rechner zu er- gen, muß das Synthese-Verfahren aus Kapitel 5 mit einem konventionellen Über- zer und Standard-Synthese-Werkzeugen kombiniert werden. Dafür bietet sich die eits in Abschnitt 2.2 (Seite 10) angegebene Grundstruktur an: Der Übersetzer t ein sequentielles Programm ein, wählt die Hardware-Kandidaten automatisch und führt die Pipeline-Synthese durch. Dann wird der Software-Anteil um An- sungen erweitert, die die RE konfigurieren, zur Laufzeit zwischen Hardware- und tware-Implementierung auswählen und gegebenenfalls die lohnenden Koprozes- n auf der RE aufrufen. Die ganze Anwendung wird also vom Wirt gesteuert.

ildung 6.2 zeigt die detaillierte Gesamtstruktur des Übersetzers: Dabei entspre- n Rechtecke den Dateien oder Datenstrukturen, die Programme oder Schaltungen räsentieren, und Ovale den Algorithmen, Werkzeugen oder Programmteilen, die e Repräsentationen (entlang den durchgezogenen Pfeilen) transformieren. Im ein- en wird das Eingabeprogramm zunächst durch ein konventionelles Übersetzer- ntend in Zwischencode übersetzt. Dieser wird im rechten Arm des Diagramms Funktionen eines Laufzeitsystems, die die Konfiguration und Ansteuerung der vornehmen (siehe Abschnitt 6.3.2), instrumentiert. Schließlich erzeugt ein kon- ionelles Backend ein Maschinenprogramm, das die FPGA-Konfigurationen auf

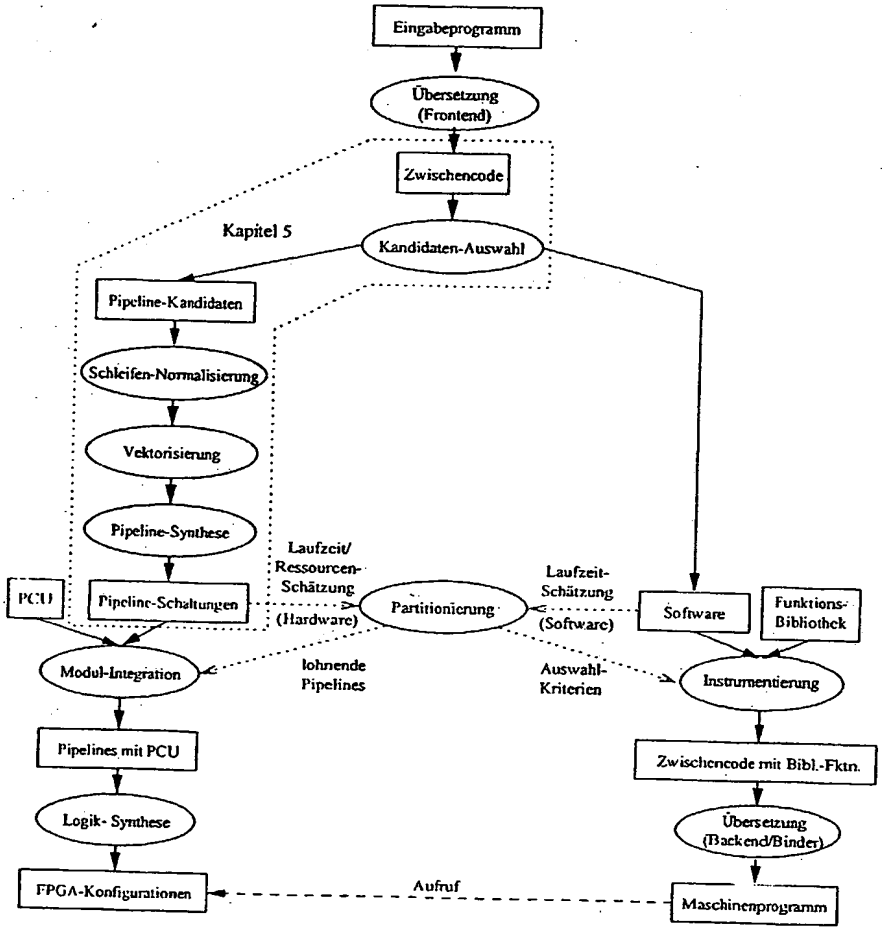


Abbildung 6.2: Gesamtstruktur des Übersetzers

ruft. Der linke Arm des Diagramms erzeugt den Hardware-Anteil: Die hellgrau unterlegten Teile entsprechen den in Kapitel 5 besprochenen Schritten von der Kandidaten-Auswahl bis zur Pipeline-Synthese. Jede lohnende Pipeline wird mit einer Pipeline-Steuerungseinheit kombiniert und durch Standard-Werkzeuge in eine FPGA-Konfiguration (Bitstrom) transformiert (siehe Abschnitt 6.3.1).

Die Hardware/Software-Partitionierung (siehe Abschnitt 6.4) entscheidet schließlich anhand von Software- und Hardware-Schätzungen einerseits, für welche Pipelines sich die Erzeugung einer FPGA-Konfiguration überhaupt lohnt, und andererseits,

nter welchen Umständen die Hardware-Implementierung zur Laufzeit tatsächlich erwendet wird, um eine maximale Beschleunigung der ganzen Anwendung zu erhalten. Sie beeinflußt also die Pipeline-Auswahl auf der Hardware-Seite und die nstrumentierung auf der Software-Seite. Die gepunkteten Pfeile in Abbildung 6.2 euten an, daß nur Informationen *über* Programm- oder Schaltungs-Komponenten usgetauscht werden, nicht aber die Komponenten selbst.

Die Struktur der Abbildung 6.2 ähnelt einem Entwurfssystem im Hardware/Software-Codesign. Jedoch erzeugen wir hier keine getrennten Hardware- und Software-Beschreibungen, sondern ein kombiniertes Maschinenprogramm, das die FPGA-Konfiguration enthält und zur Laufzeit die RE automatisch konfiguriert. Außerdem wird im Gegensatz zum Codesign die Aufteilung in Hardware- und Software-Anteile nicht statisch festgelegt. Vielmehr wird für jeden Pipeline-Kandidaten eine Software- *und* eine Hardware-Implementierung erzeugt, aus denen erst zur Laufzeit nach den n Abschnitt 6.4 entwickelten Kriterien ausgewählt wird.

6.3 Ansteuerung der Koprozessoren

Wir haben bisher die Synthese der Pipeline-Koprozessoren unabhängig von dem Software-Anteil einer Anwendung betrachtet. Damit die Koprozessoren aber in der oben angegebenen Weise von einem Programm aufgerufen werden können, muß die Ansteuerung der Hardware automatisch in die Software integriert werden. Dies betrifft zunächst die konkrete Einbettung der Pipelines in eine lauffähige Schaltung: Diese *Hardware-Integration* wird in Abschnitt 6.3.1 durch die Entwicklung einer für alle Programme einsetzbaren *Pipeline-Steuerungseinheit* erreicht. Danach erfolgt in Abschnitt 6.3.2 die *Software-Integration* durch die Instrumentierung des erzeugten Maschinenprogramms mit zusätzlichen Instruktionen, die die Koprozessoren über die Steuerungseinheit ansprechen.

6.3.1 Pipeline-Steuerungseinheit und Hardware-Integration

Wie in Kapitel 5 bereits erwähnt wurde, können die synthetisierten Pipeline-Schaltungen alleine noch keine Berechnungen durchführen: Sie benötigen dazu externe Takt- und Steuersignale. Außerdem müssen Eingaben und Ausgaben zum richtigen Zeitpunkt an den Eingängen bereitgestellt und von den Ausgängen gelesen werden. Diese Aufgaben — die Verdrahtung der Pipeline mit dem lokalen Speicher und mit der Schnittstelle zum Wirt sowie die Steuerung der Pipeline — werden von einer neuartigen Schaltungskomponente, der *Pipeline-Steuerungseinheit* (engl. pipeline control unit, PCU) übernommen. Da die Ansteuerung aller Pipelines ähnlich ist, kann *eine* PCU für alle Programme manuell entworfen und optimiert werden. Diese generische Komponente wird dann mit der Pipeline-Schaltung kombiniert, um eine funktionsfähige FPGA-Konfiguration zu erzeugen.

Pipeline-Steuerungseinheit

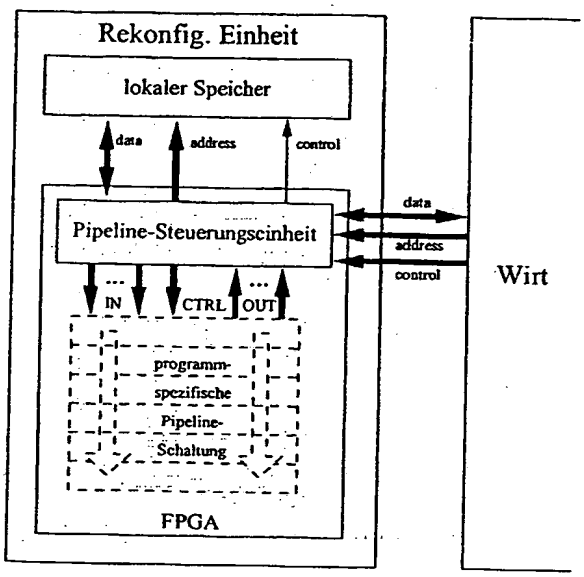


Abbildung 6.3: RE mit Pipeline-Steuerungseinheit

Aus den obengenannten Anforderungen ergibt sich folgende Schnittstelle der Steuerungseinheit: Sie verbindet die Pipeline mit den Anschlüssen des FPGA, die wiederum mit dem lokalen Speicher, mit der Schnittstelle zum Wirt und mit einem oder mehreren Taktsignalen verknüpft sind. Abbildung 6.3 zeigt diese äußere Sicht, die Einbettung der PCU in einem SP-Rechner.

Die Steuerungseinheit trennt also die Pipeline von der externen Schnittstelle. Dies erleichtert die Portierung des Übersetzers, da die Pipeline-Synthese so von der Maschinenarchitektur (also von der Schnittstelle zum Wirt und den Details der Ansteuerung des lokalen Speichers) unabhängig ist. Die einzige Abhängigkeit besteht darin, daß die generierte Schaltung die aus der Speicheransteuerung abgeleitete maximale Signalverzögerung nicht überschreiten darf. Bei einer Portierung muß also nur die PCU, die die maschinenspezifischen Details kapselt, manuell an die neue Architektur angepaßt werden. Die Anpassung ist jedoch nur einmal pro Architektur (für alle Programme) nötig.

Die Schnittstelle erlaubt der PCU, Daten zwischen Wirt, lokalem Speicher und Pipeline auszutauschen. Welche Funktionen dabei im einzelnen notwendig sind, um die Ansteuerung der Koprozessoren zu automatisieren, ergibt sich aus den in den Abschnitten 5.2.2 und 5.2.3 definierten Phasen des Pipeline-Ausführungsmodells: In den Phasen LOAD und STORE vor bzw. nach der Pipeline-Ausführung wer-

ı Daten zwischen Wirt und RE übertragen. Dabei werden skalare Daten direkt in GA-Register geschrieben bzw. skalare Ergebnisse direkt zurückgelesen und Felder Ganzes in einen zusammenhängenden Adreßbereich des lokalen Speichers kopiert x. von dort in den Hauptspeicher zurückkopiert. Und in den Phasen FILL und EC wird die Pipeline ausgeführt. Diese Phasen werden zwar vom Wirt gestartet, nen dann aber selbständig ablaufen, da während der Berechnung nur auf Daten lokalen Speicher zugegriffen wird.

muß nun eine Steuerungseinheit entworfen werden, die es ermöglicht, diese Funkn vom Software-Anteil der Anwendung gesteuert durchzuführen. Deshalb arıtet die PCU in mehreren Modi, die durch Steuerwörter vom Wirt ausgewählt rden können. Die Steuerwörter setzen entsprechende Register und Flags in der U, die die gewählte Funktion realisieren.

e Ausführung der Pipeline ist die aufwendigste Funktion, da die im lokalen RAM speicherten Daten so aufbereitet werden müssen, daß alle Vektor-Eingaben gleichtig an den Pipeline-Ports anliegen und alle Ausgaben gleichzeitig gelesen werden. es ist aber nicht direkt möglich, weil auf der RE nur eine Speicherbank vorhanden . Zu jedem Zeitpunkt kann also nur ein Schreib- oder Lesezugriff stattfinden. In serer Schaltung *simuliert* die PCU deshalb gleichzeitige Speicherzugriffe, indem sie erst alle Eingabewerte nacheinander vom Speicher liest und in PCU-Registern zwibenpuffert, und dann alle Pipeline-Ausgaben in den Speicher zurückschreibt. Ein peline-Takt besteht also aus mehreren internen Zyklen, in denen die Schreib- und sezugriffe nacheinander erfolgen. In Abbildung 6.4 sind die dafür nötigen internen mponenten der PCU (ohne Steuerwerk) dargestellt: Die Speicher-Basisadressen ier Ports (BASEREG i) sowie die zugehörigen Schrittweiten (STRIDE i) werı vor der Pipeline-Ausführung vom Wirt in die Register geladen. Während der sführung werden sie dann in einem Pipeline-Takt nacheinander für alle Ports auswählt und mit dem Schleifenzähler-Register (LOOP COUNTER) verknüpft. So ird für jeden Port die aktuelle Adresse im lokalen Speicher generiert.

ie Dauer der internen Zyklen hängt von der Geschwindigkeit der verwendeten M-Bausteine und von den verfügbaren Taktsignalen ab, ist also für eine Architur fest. Dagegen wird die Zahl und Abfolge dieser Zyklen in einem Pipeline-Takt n den verwendeten Vektor-Ports bestimmt (vgl. Abschnitt 5.2.3). Deshalb muß die CU durch ein Steuerwort vom Wirt mit der Anzahl der verwendeten Eingabe- und usgabe-Ports parametrisiert werden.

ie richtige Zahl der Pipeline-Takte in den Phasen FILL und EXEC wird folgenrmaßen vom Steuerwerk der PCU generiert: Beim Start der Ausführung wird die ktuelle Schleifenlänge durch ein Steuerwort vom Wirt in einem (in Abbildung 6.4 cht gezeigten) PCU-Register gespeichert. Außerdem wird der LOOP COUNTER it der negierten Länge des längsten Eingabe-Schieberegisters initialisiert und mit m Beginn der FILL-Phase in jedem Pipeline-Takt inkrementiert. Hat der Zähler n Wert Null erreicht, sind alle Eingabe-Schieberegister gefüllt und die EXEChase beginnt. Dann signalisiert das Steuersignal VALID gültige Werte in der ersten ipeline-Stufe (vgl. Abschnitt 5.2.3). Bei FOR-Schleifen wird der LOOP COUNTER

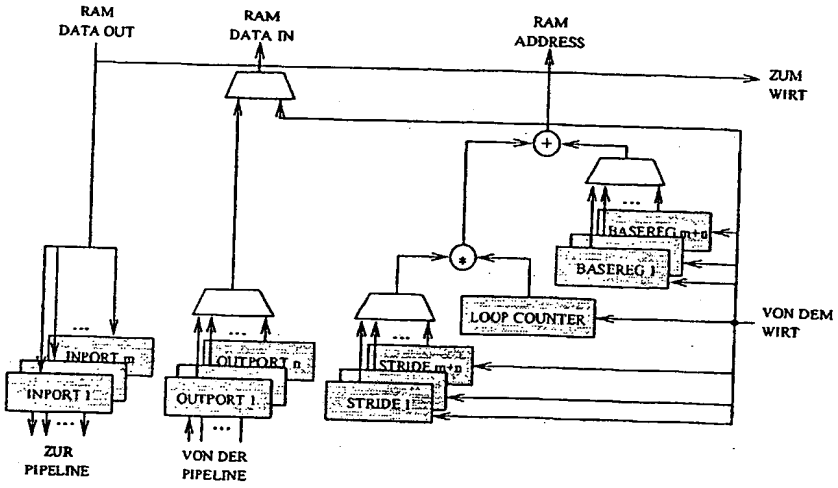


Abbildung 6.4: Interner Aufbau der Pipeline-Steuerungseinheit

solange weiter inkrementiert, bis er die aktuelle Schleifenlänge erreicht. Dann wird VALID zurückgesetzt. Die Pipeline wird aber solange weitergetaktet, bis auch das Steuersignal VALID_OUT von der Pipeline zurückgesetzt wird. So wird das Ende der EXEC-Phase verzögert, bis die Pipeline leergelaufen ist. Bei WHILE-Schleifen wird VALID_OUT und das Ende der EXEC-Phase jedoch durch das in der Pipeline erzeugte Abbruch-Signal gesteuert.

Die Übertragung der Felder vom und zum Wirt in den Phasen LOAD und STORE verwendet ebenfalls die Speicher-Ansteuerung mit dem LOOP COUNTER und einem Basisregister. Die Steuerung ist ähnlich wie bei EXEC, jedoch wird die Inkrementierung des LOOP COUNTER durch die Daten-Übertragungen vom Wirt ausgelöst. Im Gegensatz dazu ist bei der Übertragung skalarer Daten keine komplizierte Steuerung notwendig: Die zu lesenden oder zu schreibenden Register werden vom Wirt einfach über entsprechende Auswahlsignale, die an die Pipeline-Schaltung weitergegeben werden, angesprochen.

Generierung der FPGA-Konfigurationen

Die Steuerungseinheit muß nun mit der Pipeline-Schaltung zu einer vollständigen FPGA-Konfiguration kombiniert werden: Dazu werden die Operatoren jeder Pipeline-Schaltung zuerst durch entsprechende Module einer Bibliothek für die verwendete FPGA-Familie instanziert. Dann wird die resultierende Netzliste der Schaltung auf Register-Transfer-Ebene mit der manuell erzeugten Netzliste der Pipeline-

Steuerungseinheit kombiniert (*Modul-Integration*).[1] Da auch die Zahl der zur Implementierung der Bibliotheks-Module nötigen Logik-Blöcke bekannt ist, können wir die Gesamtzahl der für die ganze Netzliste nötigen Blöcke bestimmen und entscheiden, ob das vorhandene FPGA für diese Pipeline ausreicht. Ist dies nicht der Fall, kann die Schleife nicht in Hardware realisiert werden.[2] Schließlich erzeugen Standard-Werkzeuge einen Konfigurations-Bitstrom, der in das auf dem Wirt laufende Programm eingebunden und zur automatischen Konfigurierung des FPGA verwendet wird (*Logik-Synthese*).

Die Durchführung der Logik-Synthese (einschließlich Plazierung und Verdrahtung) kann für ein FPGA auch auf einem schnellen Rechner mehrere Stunden dauern. Dagegen dauert die Vektorisierung und Pipeline-Synthese nicht länger als die anderen Übersetzer-Phasen (maximal wenige Sekunden). Es ist also sinnvoll, die Logik-Synthese nur in lohnenden Fällen durchzuführen. Deshalb wird bereits bei der Kombination der Netzlisten geprüft, ob die Hardware für eine Pipeline ausreicht. Außerdem schätzt die erste Stufe der Partitionierung (Abschnitt 6.4) nach der Pipeline-Synthese ab, ob sich der Einsatz eines Koprozessors voraussichtlich lohnt. Nur in diesem Fall wird die Logik-Synthese durchgeführt. Leider kann es jedoch auch vorkommen, daß eine Schaltung, deren Logik-Blöcke plaziert werden konnten, nicht verdrahtet werden kann. In diesem Fall muß die lange Synthese-Zeit in Kauf genommen werden, da keine zufriedenstellenden Verfahren zur Vorhersage der Verdrahtungsergebnisse zur Verfügung stehen.

6.3.2 Software-Integration

Wir haben gezeigt, wie die Pipeline-Schaltungen durch eine PCU gesteuert werden. Nun muß aber das auf dem Wirt laufende Programm wiederum die Funktionen der PCU steuern. Diese Software-Integration geschieht durch die Instrumentierung des Zwischencodes mit manuell entwickelten Bibliotheksfunktionen. Sie initialisieren die RE, konfigurieren das FPGA, veranlassen die Übertragung von Daten und rufen den Pipeline-Koprozessor auf, indem sie Steuerworte zur Auswahl der richtigen Funktion an die RE senden und die PCU-Register setzen. Dabei wird die Software-Implementierung nicht durch einen Koprozessor-Aufruf *ersetzt*, sondern um ihn *ergänzt*. Die Auswahl zwischen der Software- und der Hardware-Implementierung erfolgt dann zur Laufzeit nach den Kriterien, die die zweite Stufe des Partitionierungs-Verfahrens (Abschnitt 6.4) festlegt.

Da die RE nicht auf den Hauptspeicher zugreifen kann, müssen alle verwendeten Variablen in der RE gespeichert werden. Für die skalaren Variablen wurden dafür

[1] Es wäre auch möglich, mehrere kleine Pipelines mit einer PCU zu einer Konfiguration zu kombinieren und zur Laufzeit jeweils eine Pipeline auszuwählen. Dann könnte häufig ohne Umkonfigurierung zwischen den Pipelines gewechselt werden. Auf diese Erweiterung gehen wir in Abschnitt 9.3 ein.

[2] Durch Aufteilen des Schleifen-Rumpfes in mehrere Einzel-Schleifen könnten dennoch Pipelines für die Schleife erzeugt werden. Diese Erweiterung wird in Abschnitt 9.2 betrachtet.

bereits FPGA-Register in der Pipeline-Schaltung angelegt. Jedoch müssen den Feldern vor der Instrumentierung Teilbereiche des lokalen Speichers zugeordnet werden. Da bei einer Umkonfigurierung der RE nach unserem SP-Rechner-Modell auch der lokale Speicher gelöscht wird, wird für jede Pipeline eine eigene Allokationstabelle angelegt. Sie legt für alle in der Pipeline verwendeten Felder die Basisadresse und die Länge des Feldes fest. Wegen der gleichen Wortlänge auf Wirt und RE sind die Längen und auch die relativen Verschiebungen (Offsets) zur Berechnung der Basisadressen von Teilfeldern gleich wie im Hauptspeicher. Deshalb kann das ganze Feld auch wortweise zwischen Hauptspeicher und lokalem Speicher kopiert werden. Reicht der lokale Speicher allerdings nicht für alle Felder aus, kann die Pipeline nicht implementiert werden.[3]

Die Instrumentierung für einen Koprozessor-Aufruf beginnt mit einer Funktion, die die RE initialisiert und das FPGA konfiguriert. Da dies sehr lange dauert, sollte die Funktion vor einer Umkonfigurierung überprüfen, ob das FPGA bereits richtig konfiguriert ist.

Danach müssen für alle Variablen, die im Software-Anteil definiert und vom Koprozessor benutzt werden, Funktionen zum Kopieren der Variablen zur RE eingefügt werden. Wird der Koprozessor nur einmal aufgerufen, sind dies alle Eingaben (Menge INPUT aus Abschnitt 5.2.2) sowie die Felder aus der Menge OUTPUT. Die letzteren müssen kopiert werden, da die in OUTPUT vorkommenden Felder — im Gegensatz zu den skalaren Variablen — nur partiell definiert werden. Denn es werden nicht an alle Feldelemente notwendigerweise neue Werte zugewiesen. Für die nicht definierten Elemente gelten folglich die im Software-Anteil definierten Werte. Diese müssen im lokalen Speicher der RE verwendet werden, um keine undefinierten Werte zu erhalten. Denn diese würden bei der Rückübertragung eines Feldes nach dem Pipeline-Aufruf die gültigen Werte im Hauptspeicher überschreiben. Da eine genaue Analyse der in einer Schleife definierten Feldelemente jedoch meist nicht möglich ist und eine selektive Übertragung einzelner Feldelemente die Datenübertragung komplizierter machen würde, muß das ganze Feld als benutzt betrachtet und zum lokalen Speicher kopiert werden.

Nach diesen Kopier-Funktionen wird der Zwischencode mit dem Aufruf des Koprozessors instrumentiert. Diese Funktion stoppt das Programm auf dem Wirt, bis die Pipeline-Ausführung beendet ist. Danach werden die Funktionen zum Rückkopieren der Ergebnisse zum Hauptspeicher eingefügt. Dies betrifft alle vom Koprozessor definierten und im Software-Anteil benutzten Variablen. Wird der Koprozessor nur einmal aufgerufen, sind dies die nach dem Aufruf *lebendigen* Variablen aus OUTPUT. Wir führen also eine *Lebendigkeits-Analyse* durch, damit die im Software-Anteil nicht mehr benötigten Variablen nicht zurückkopiert werden.

Die notwendigen Kopier-Funktionen sind jedoch schwerer zu bestimmen, wenn ein

[3] Um die Schleifen trotzdem bearbeiten zu können, wären Transformationen nötig, die Teilfelder getrennt in die RE laden und bearbeiten. Eine solche Erweiterung wird ebenfalls in Abschnitt 9.2 betrachtet.

prozessor mehrmals aufgerufen wird. Denn Variablen, die nur im Koprozessor hrfach bearbeitet werden, sollten nicht zum Hauptspeicher und anschließend unändert wieder zur RE kopiert werden. In diesem Fall hängen die benötigten pier-Funktionen von der Hardware/Software-Partitionierung ab. Die erforderli- Plazierung der Funktionen wird deshalb in Abschnitt 6.4 erläutert.

ch der Instrumentierung wird der Zwischencode schließlich in ein Maschinenprmm übersetzt. Zu dem Programm werden die FPGA-Bitströme der erzeugten nfigurationen gebunden, so daß zur Laufzeit ein Bitstrom ausgewählt, das FPGA nfiguriert und die richtige Pipeline verwendet werden kann.

4 Hardware/Software-Partitionierung

Kapitel 5 und in den vorangehenden Abschnitten dieses Kapitels wurden die peline-Schaltungen, die Pipeline-Steuerungseinheit und das zugrundeliegende Mainenmodell für einen kombinierten Übersetzer vorgestellt. Diese stellen die Randdingungen dar, unter denen für jeden Hardware-Kandidaten zwischen einer ftware-Implementierung und einer Hardware-Implementierung ausgewählt werden ß.

el dieser Hardware/Software-Partitionierung ist es, ein Programm so in Hardware- d Software-Anteile aufzuteilen, daß seine Gesamtlaufzeit minimiert wird. Die Verren des Hardware/Software-Codesign legen dazu *statisch*, also zur Übersetzungsit, Hardware- und Software-Komponenten fest (vgl. Abschnitt 4.2). Dann kann rch eine DEF/USE-Analyse der Variablen genau bestimmt werden, welche Daten nn zum Koprozessor und zurück kopiert werden müssen (siehe Anhang B). Audem wird nur *eine* Schaltung erzeugt, da entweder (nicht konfigurierbare) ASICs rgestellt sind oder aus anderen Gründen keine Umkonfigurierung zur Laufzeit öglich ist. Eine solche statische Partitionierung ist aber nur dann sinnvoll, wenn r die Anwendung aussagekräftige und gleichartige Profiling-Daten vorliegen. Dies . bei eingebetteten Systemen häufig der Fall. Für eine statische Partitionierung richt bei diesen Systemen auch, daß oft nur ein kleiner Speicher für den Hauptozessor vorgesehen ist. In diesem Fall würde die doppelte Implementierung vieler nktionen in Hardware *und* Software zusätzliche Kosten verursachen.

agegen ist die Bereithaltung sowohl einer Software- als auch einer Hardwareplementierung in einem SP-Rechner problemlos möglich. Dies ist gerade bei unsen Pipeline-Schaltungen sinnvoll, da die Hardware-Beschleunigung *datenabhängig* t: Sie wird von der aktuellen Schleifenlänge bestimmt. Beispielsweise hängt die eschleunigung für die Bildverarbeitung aus Kapitel 2 stark von der Größe des Einbebildes ab; nicht für alle Bildgrößen lohnt sich die Verwendung einer Hardwareipeline. Die aktuelle Eingabegröße läßt sich aber prinzipiell nicht durch Profiling mitteln. Wir entwickeln deshalb ein neuartiges zweistufiges Partitionierungsverfahn, das in der ersten Stufe zur Übersetzungszeit anhand grober Schätzungen für die ftware- und Hardware-Laufzeit bestimmt, für welche Schleifen sich die Synthese

eines Pipeline-Koprozessors prinzipiell lohnt. Danach ermittelt die zweite Stufe zur Laufzeit genauere Schätzungen aus den Eingabedaten und wählt *dynamisch* eine Implementierung aus. Da sich mit der Auswahl auch die für eine korrekte Ausführung notwendigen Kopier-Funktionen ändern, müssen auch diese entsprechend angepaßt werden. Um die Laufzeit-Kosten für diese dynamische Auswahl in Grenzen zu halten, müssen wir einen Kompromiß zwischen der erreichten Flexibilität und der Einfachheit der Instrumentierung finden.

Zusätzlich ist bei SP-Rechnern auch eine *dynamische Rekonfigurierung* zur Laufzeit sinnvoll, falls damit die Gesamtlaufzeit verkürzt wird. Denn ein Programm kann aus mehreren Phasen bestehen, die jeweils für sich durch Hardware-Pipelines beschleunigt werden können und zwischen denen sich eine Umkonfigurierung lohnt. Wir erweitern die Laufzeit-Auswahl deshalb um eine ebenfalls neuartige Laufzeit-Rekonfigurierung, die die Phasen, zwischen denen umkonfiguriert wird, automatisch bestimmt.

Eine inkrementelle Hardware-Schätzung (vgl. Abschnitt 4.2) verschiedener Schaltungen ist für diese Partitionierung nicht notwendig, da nur jeweils eine Pipeline in einer Konfiguration realisiert wird. Die in Abschnitt 6.3.1 erwähnte Überprüfung des Hardware-Bedarfs reicht aus.

6.4.1 Schätzung der Ausführungszeiten

Für beide Stufen der Partitionierung ist eine Schätzung der Software- und Hardware-Ausführungszeiten jedes Hardware-Kandidaten L nötig. Wir diskutieren deshalb in diesem Abschnitt zunächst diese Schätzungen: Die Hardware-Ausführungszeit setzt sich aus der *reinen Hardware-Ausführungszeit* $T_{HW}(i)$ des i-ten Koprozessor-Aufrufs, der Konfigurierungs- und der Kommunikationszeit zusammen. $T_{HW}(i)$ ist einfach und präzise vorherzusagen: Da nach dem Füllen in jedem Pipeline-Takt eine Schleifeniteration berechnet wird, kann sie folgendermaßen aus der aktuellen Schleifenlänge $l(i)$, der nach den Formeln (5.12) und (5.13) berechneten Taktperiode T_C und einer konstanten Zeit c zur Ansteuerung und zum Füllen der Pipeline berechnet werden:

$$T_{HW}(i) = l(i) \times T_C + c \tag{6.1}$$

Die *Konfigurierungszeit* T_{DL}[4] ist in unserem SP-Rechner-Modell fest, da die ganze RE nur komplett rekonfiguriert werden kann. Um aber die *Kommunikationszeit* T_{Komm} abschätzen zu können, müssen wir eine vereinfachende Annahme treffen: Wir gehen davon aus, daß alle in einem Koprozessor verwendeten Variablen im Software-Anteil zusammen referenziert werden. Dies bedeutet, daß für eine Schleife L immer ein ganzer Datensatz D(L) übertragen wird. T_{Komm} entspricht dann der Gesamtzeit zur Übertragung aller benötigten Variablen zur RE und zur Rückübertragung aller Ergebnisse zum Hauptspeicher.[5]

[4] engl. download time

[5] Wir gehen auch davon aus, daß T_{Komm} von den aktuellen Eingabedaten unabhängig ist. Dies

die Software-Ausführungszeit $T_{SW}(i)$ bestimmen wir die Dauer T_P einer Schlei-
teration auf dem Hauptprozessor als Summe der Laufzeiten der einzelnen Befehle
Zwischencode. Bei bedingten Anweisungen werden außerdem Annahmen über die
ufigkeit der Programmzweige gemacht und die Zweige entsprechend gewichtet. Es
jedoch schwierig, bei dieser Analyse Übersetzer-Optimierungen und Laufzeitef-
te (etwa Cache-Hits und -Misses) zu berücksichtigen. Die folgende Schätzung der
tware-Laufzeit für die i-te Schleifenausführung ist deshalb recht grob:

$$T_{SW}(i) = l(i) \times T_P \tag{6.2}$$

ı die späteren Berechnungen zu vereinfachen, definieren wir schließlich die *Be-
leunigung einer einzelnen Schleifeniteration* $\Delta_{SW/HW}$ (ohne Berücksichtigung der
mmunikations- und Konfigurierungszeiten) folgendermaßen:

$$\Delta_{SW/HW} = T_P - T_C \tag{6.3}$$

$_{SW/HW}$ ist jedoch recht ungenau, da sich die Instrumentierung und die Datenüber-
gungen auch auf die Anwendbarkeit der Übersetzer-Optimierungen und auf die
ufzeiteffekte auswirken. Leider ist es aber kaum möglich, diese Auswirkungen
rherzusagen.

lls für eine Schleife L $\Delta_{SW/HW} < 0$ gilt, die Software-Ausführung einer Iteration
so schneller als ein Pipeline-Takt ist, wäre die Hardware auch ohne jegliche Zusatz-
sten langsamer als die Software. Es lohnt sich dann keinesfalls, einen Koprozessor
synthetisieren.

.4.2 Bestimmung der lohnenden Pipelines

ie erste Stufe der Partitionierung bestimmt, für welche Hardware-Kandidaten sich
e Generierung einer FPGA-Konfiguration trotz der Zusatzkosten voraussichtlich
hnt. Es muß also geprüft werden, ob eine Phase des Programmlaufs, in der nicht
nkonfiguriert wird, durch eine Pipeline beschleunigt werden kann. Da der genaue
blauf des Programms jedoch nicht bekannt ist, wissen wir im voraus auch nicht,
ann rekonfiguriert und wie oft kommuniziert werden muß. Die Gesamtlaufzeiten
it und ohne Pipeline müssen also *geschätzt* werden.

eshalb wird zunächst analysiert, wie oft ein Kandidat L ohne zwischenzeitli-
ne Kommunikation aufgerufen wird. Dazu suchen wir im Programm die äußerste
chleife K, die L einschließt und in welcher der Datensatz D(L) (außerhalb von L)
icht referenziert, d. h. definiert oder benutzt, wird. Denn in dieser Schleife fällt
$_{Komm}$ nur einmal an. Im ungünstigsten Fall ist K = L.[6] In dem Beispielprogramm

t formal korrekt, da die Größe der Felder des Eingabeprogrammes konstant ist. Jedoch wäre
ei nur teilweise gefüllten Feldern die Übertragung des ganzen Feldes eigentlich nicht notwendig,
odurch auch T_{Komm} datenabhängig würde.

[6]Auch wenn D(L) in einer äußeren Schleife nur manchmal (in einer bedingten Anweisung)

aus Kapitel 5 (Abbildung 5.1) ist K die äußere Schleife L1, und in den Bildverarbeitungsprogrammen (Abschnitt 2.1) entspricht K den Schleifen über alle Bildzeilen.

Da nicht genau bekannt ist, wie oft L in der Schleife K ausgeführt wird und wieviele Iterationen die einzelnen Ausführungen der Schleife L haben, müssen die Ausführungszahl k (von L in K) und die durchschnittliche Schleifenlänge l durch eine statische Analyse der Ausführungshäufigkeit von Schleifen und bedingten Anweisungen geschätzt werden. (Diese Analysen können auch Annotationen des Programmierers verwenden, um genauere Schätzungen zu erhalten.) Dann ergeben sich folgende geschätzte Ausführungszeiten für K:

$$T_{K,SW} = k \times l \times T_P \tag{6.4}$$

$$T_{K,HW} = k \times (l \times T_c + c) + T_{Komm} \tag{6.5}$$

T_{Komm} fällt also für alle Ausführungen nur einmal an.

Bei dieser Analyse wurde aber T_{DL} noch nicht berücksichtigt. Im nächsten Schritt wird deshalb die äußerste Schleife P, die K einschließt und in der kein anderer Kandidat liegt, gesucht. (Im ungünstigsten Fall ist P = K.) Innerhalb dieses Programm-Abschnittes P muß die RE nicht umkonfiguriert werden, T_{DL} fällt also nur einmal an. In unseren Beispielen entspricht P dem ganzen Programm, da wir immer nur eine vektorisierbare Schleife betrachten. Abbildung 6.5 verdeutlicht das Verhältnis der Schleifen P, K und L für verschiedene Kandidaten in einem Programm.

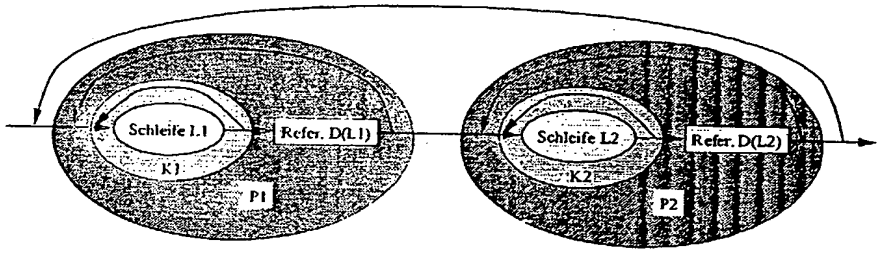


Abbildung 6.5: Beispiel für die Schleifen P und K

Wir analysieren wiederum statisch, wie oft K in P durchschnittlich ausgeführt wird. Das Ergebnis sei p, wodurch sich die folgenden Zeitschätzungen für K ergeben:

$$T_{P,SW} = p \times k \times l \times T_P \tag{6.6}$$

$$T_{P,HW} = p \times (k \times (l \times T_c + c) + T_{Komm}) + T_{DL} \tag{6.7}$$

$$= p \times k \times l \times T_c + p \times k \times c + p \times T_{Komm} + T_{DL} \tag{6.8}$$

referenziert wird, gehört diese Schleife nicht mehr zu K. In diesem Fall erhalten wir eine zu pessimistische Abschätzung, da L tatsächlich öfter als geschätzt ausgeführt werden kann, ohne daß D(L) referenziert wird.

Nun können wir bestimmen, in welchen Fällen die Software-Ausführung von P länger als die Hardware-Ausführung (einschließlich Rekonfigurierung und Kommunikationen) dauert. Die Synthese einer FPGA-Konfiguration lohnt sich also, wenn folgende Ungleichung gilt:

$$T_{P,SW} > T_{P,HW} \tag{6.9}$$

$$\Leftrightarrow \quad p \times k \times l \times T_P > p \times k \times l \times T_c + p \times k \times c + p \times T_{Komm} + T_{DL} \tag{6.10}$$

$$\Leftrightarrow \quad p \times k \times l \times \Delta_{SW/HW} - p \times k \times c > p \times T_{Komm} + T_{DL} \tag{6.11}$$

$$\Leftrightarrow \quad l \times \Delta_{SW/HW} - c > \frac{T_{Komm}}{k} + \frac{T_{DL}}{p \times k} \tag{6.12}$$

Die Darstellung in Ungleichung (6.12) verdeutlicht, daß sich ein Hardware-Koprozessor genau dann lohnt, wenn die durchschnittliche Beschleunigung $l \times \Delta_{SW/HW} - c$ für eine Schleifenausführung größer als die *amortisierten* Konfigurierungs- und Kommunikationskosten für alle Ausführungen in P ist.

Leider beeinflußt diese Auswahl der lohnenden Pipelines aber ihre eigenen Voraussetzungen: Denn durch weniger ausgewählte Konfigurationen kann sich für die verbliebenen Konfigurationen der Abschnitt P, in dem nicht umkonfiguriert werden muß, vergrößern. Also wird auch p größer, die amortisierten Konfigurierungskosten werden kleiner, und der Vorteil einer Hardware-Implementierung steigt. Zwei Pipelines, die Ungleichung (6.12) nicht erfüllen, könnten sich also jeweils einzeln lohnen. Da es schwierig ist, diese gegenseitigen Abhängigkeiten global zu analysieren, können wir folgende Heuristik verwenden: Die Kandidaten werden schrittweise entfernt, damit sich nicht alle gegenseitig ausschließen. Dabei wird zuerst der ungünstigste Kandidat (mit dem größten Wert für $T_{P,HW} - T_{P,SW}$) entfernt, und für die anderen Kandidaten werden die Schleifen P sowie ihre Ausführungszahlen p neu bestimmt. Dies wird so lange fortgesetzt, bis nur noch lohnende Kandidaten übrigbleiben.

Da die ganze Analyse auf recht ungenauen Schätzungen beruht, sollte dem Programmierer auch alternativ die Möglichkeit gegeben werden, durch Annotationen auszuwählen, für welche Schleifen FPGA-Konfigurationen erzeugt werden sollen. Auch wenn diese Entscheidung durch den Programmierer getroffen wird, kann die im nächsten Abschnitt angegebene Heuristik zur automatischen Laufzeit-Auswahl eingesetzt werden.

6.4.3 Laufzeit-Auswahl

Nach der im vorhergehenden Abschnitt vorgenommenen Analyse können wir davon ausgehen, daß sich eine Rekonfigurierung bei jeder Ausführung einer Schleife P, für die eine FPGA-Konfiguration erzeugt wurde, lohnt. Nach Bedingung (6.12) ergibt sich eine durchschnittliche Beschleunigung, wenn für den Kandidaten L immer die Pipeline aufgerufen wird. Wir könnten also statisch partitionieren, indem wir vor der Schleife P die Konfigurierungs-Funktion einfügen, vor und nach der Schleife K die

- Datensätze D(L) übertragen und statt der Schleife L immer einen Pipeline-Aufruf einfügen.

Falls eine genauere Schätzung möglich ist, soll aber dynamisch zur Laufzeit entschieden werden, ob die aktuelle Schleifenausführung schneller in Software oder in Hardware erfolgt. Außerdem soll das FPGA nur dann konfiguriert werden, wenn es tatsächlich benutzt wird. Deshalb wird im folgenden die zweite Stufe der Partitionierung, die *Laufzeit-Auswahl*, entwickelt.

Wir könnten vor jeder Schleifen-Ausführung prüfen, ob $T_{SW}(i) > T_{HW}(i)$ gilt. Dies könnte aber bei abwechselnder Ausführung in Software und Hardware zu vielen zusätzlichen Kopier-Operationen führen, die den Vorteil der individuellen Auswahl zunichte machen.

Deshalb wird einmal *für jeden Datensatz* (also für jede Ausführung der Schleife K) geprüft, ob für alle Schleifenausführungen in K die Software- oder die Hardware-Implementierung günstiger ist. Dafür werden für den aktuellen Datensatz exakte Werte oder genauere Schätzungen der durchschnittlichen Schleifenlänge l und der Ausführungszahl k benötigt. Exakte Werte können bestimmt werden, wenn k und l für alle Schleifenausführungen in K gleich sind und direkt von dem Datensatz abhängen. Dies ist etwa in unserem Beispielprogramm (Abbildung 5.1) mit $k =$ `MAXITER` und $l =$ `N-2` sowie in den Bildverarbeitungsprogrammen aus Abschnitt 2.1 mit $k =$ `VERLEN-2` und $l =$ `HORLEN-2` der Fall. Andernfalls können die Ergebnisse der statischen Analyse oft durch Einsetzen der aktuellen Variablenwerte präzisiert werden.

Mit diesen aktuellen Werten von l und k wird die Hardware-Ausführung gezielt gewählt, wenn $T_{K,SW} > T_{K,HW}$ gilt:

$$
\begin{aligned}
& T_{K,SW} > T_{K,HW} && (6.13)\\
\Leftrightarrow\ & k \times l \times T_P > k \times (l \times T_c + c) + T_{Komm} && (6.14)\\
\Leftrightarrow\ & k \times (l \times \Delta_{SW/HW} - c) > T_{Komm} && (6.15)\\
\Leftrightarrow\ & l \times \Delta_{SW/HW} - c > \frac{T_{Komm}}{k} && (6.16)
\end{aligned}
$$

Wir betrachten in (6.16) also nur die amortisierten Kommunikationskosten, da wir davon ausgehen können, daß sich die einmalige Konfigurierung des FPGA vor der Schleife P lohnt. Da die Auswahl nur einmal für jeden Datensatz getroffen wird, werden in K keine weiteren Kommunikationen, die die Gesamtausführungszeit verlängern könnten, notwendig.

Für die Auswertung zur Laufzeit wird Ungleichung (6.15) verwendet: Da $\Delta_{SW/HW}$, c und T_{Komm} zur Übersetzungszeit bekannte Konstanten sind, müssen nur die veränderlichen Werte für k und l eingesetzt werden.

Durch diese Analyse werden also nicht lohnende Koprozessor-Aufrufe verhindert. Um dies auch für unnötige Konfigurierungen zu erreichen, wird eine weitere Optimie-

rung durchgeführt: Da erst dann umkonfiguriert werden sollte, wenn die Hardware-Implementierung tatsächlich gewählt wird, koppeln wir die Konfigurierung mit der Auswahl. Wir instrumentieren den Zwischencode also so, daß an den Eingängen der Regionen K die Ungleichung (6.15) überprüft wird. Gilt sie, wird — falls nötig — umkonfiguriert, kommuniziert und die Pipeline ausgeführt. Andernfalls wird die Software gewählt. Der aktuelle Konfigurationszustand der RE wird gespeichert und vor einer Rekonfigurierung überprüft, so daß in jeder Ausführung der Region P maximal einmal umkonfiguriert wird. Der zusätzliche Aufwand dieser Prüfung ist sehr klein im Vergleich zu den Kosten einer möglicherweise unnötigen Umkonfigurierung.

Abbildung 6.6 zeigt unser Beispielprogramm mit den eingefügten Instruktionen zur Laufzeit-Auswahl. Dabei sind `DELTA_SW_HW`, `C` und `T_KOMM` Konstanten, die zur Übersetzungszeit berechnet werden. Zur Laufzeit müssen also nur `N` und `MAXITER` eingesetzt werden, falls dies Variablen sind. Die Funktion `RT_DOWNLOAD` überprüft die aktuelle Konfiguration und ändert sie gegebenenfalls, und die Funktion `RT_PIPELINE` ruft die aktuelle Pipeline auf. (Sie hat tatsächlich mehr als die hier angegebenen Parameter.) Die restlichen Bibliotheksfunktionen führen die Datenübertragungen aus.

```
...
IF (MAXITER * ((N - 2) * DELTA_SW_HW - C) > T_KOMM) THEN
  RT_DOWNLOAD(1);       (* Lade Konfiguration 1 *)
  RT_PUT_VEC(X,0,N);    (* Kopiere X *)
  RT_PUT_SCAL(RAND,1); (* Kopiere RAND *)
  FOR ITER := 1 TO MAXITER DO (* L1 *)
    RT_PIPELINE(N-2,...); (* Rufe aktuelle Pipeline auf *)
    S := 1 - S;
  END;
  RT_GET_VEC(X,0,N);    (* Kopiere X zurueck *)
  RT_GET_SCAL(RAND,1); (* Kopiere RAND zurueck *)
ELSE
  FOR ITER := 1 TO MAXITER DO (* L1 *)
    ... (* Software *)
    S := 1 - S;
  END;
END
```

Abbildung 6.6: Instrumentiertes Beispielprogramm

6.5 Bewertung

In diesem Kapitel wurde ein neuer Ansatz zur kombinierten Strukturprogrammierung vorgestellt. Er ist weitgehend maschinenunabhängig und ermöglicht so den Ein-

satz des Pipeline-Synthese-Verfahrens aus Kapitel 5 auf verschiedenen SP-Rechnern.

Die aus geeigneten Schleifen des Eingabeprogramms generierten Pipeline-Koprozessoren werden von einer neuartigen generischen Schaltungskomponente, der Pipeline-Steuerungseinheit, angesteuert. So können die Koprozessoren verschiedener Programme einheitlich behandelt werden. Die Steuerungseinheit vereinfacht außerdem die Portierung des Verfahrens, da sie die maschinenspezifischen Details des SP-Rechners kapselt.

Die Funktionen der Steuerungseinheit werden wiederum von Bibliotheksfunktionen, mit denen das auf dem Wirt laufende Programm instrumentiert wird, gesteuert. Da auch die FPGA-Bitströme zum erzeugten ausführbaren Maschinenprogramm gebunden werden, kann es die RE zur Laufzeit automatisch konfigurieren und steuern.

In diesem System erzeugt ein kombinierter Übersetzer also erstmals automatisch Pipeline-Koprozessoren für geeignete Programmteile und integriert ihre Ansteuerung in den Software-Anteil der Anwendung.

Die Auswahl der lohnenden Programmteile wird durch ein neuartiges zweistufiges Hardware/Software-Partitionierungsverfahren vorgenommen. Die erste Stufe entscheidet statisch, welche Schleifen sich prinzipiell durch Hardware-Pipelines beschleunigen lassen, und die zweite Stufe wählt dynamisch zur Laufzeit — abhängig von den aktuellen Eingabedaten — die voraussichtlich schnellste Implementierung in Hardware oder Software aus und rekonfiguriert die Hardware gegebenenfalls.

Im Gegensatz zu den aus der Literatur bekannten Verfahren wählt diese Partitionierung zur Laufzeit datenabhängig zwischen Software und Hardware aus. Außerdem wird die Rekonfigurierungszeit der RE berücksichtigt. Eine Analyse bestimmt sogar automatisch die Programmphasen, zwischen denen sich eine Rekonfigurierung zur Laufzeit lohnt.

Kapitel 7

Der MODULA PIPELINE COMPILER

Dieses Kapitel präsentiert eine prototypische Implementierung der in den vorhergehenden Kapiteln vorgestellten Verfahren, den MODULA PIPELINE COMPILER. Er zeigt, daß die Integration der Vektorisierung und Pipeline-Synthese in einen Übersetzer praktikabel ist, erhebt aber nicht den Anspruch, vollständig oder abgerundet zu sein. Deshalb wurden in dieser Version des Übersetzers Funktionen weggelassen, die nebensächlich sind oder in anderen Arbeiten bereits demonstriert und dokumentiert wurden. Außerdem konnte aus Zeitgründen die automatische Partitionierung nicht mehr implementiert werden. Trotzdem können die Erfahrungen mit diesem kleinen Prototyp eine große Hilfe für eine spätere Entwicklung eines vollständigen Übersetzers für größere Architekturen sein.

Zunächst werden die verwendete Hardware-Plattform — eine Sun SPARCstation mit einer EVC1-Karte — und der MODULA-2-Übersetzer MOCKA, auf dem die Implementierung basiert, vorgestellt. Danach gehen wir auf Details des Prototypen ein, berichten über Erfahrungen und schlagen Verbesserungen für eine Neuimplementierung vor.

7.1 Die Hardware-Plattform

Als Zielplattform des MODULA PIPELINE COMPILER dient eine *Sun SPARCstation 10/40* (40 MHz SuperSPARC-Prozessor unter SunOS 4.1.3) mit der SBUS-Karte *Engineer's Virtual Computer EVC1* [TCS94, CTS95], die im nächsten Abschnitt näher vorgestellt wird. Diese Kombination wurde ausgewählt, da der EVC1 die erste kommerziell verfügbare, kleine FPGA-Einsteckkarte war, die für unsere Zwecke geeignet ist. Durch die SBUS-Schnittstelle der Karte kommt als Wirt nur eine Sun Workstation in Frage. Außerdem existiert auch eine Version des MOCKA für diese im akademischen Bereich sehr verbreiteten Workstations.

7.1.1 Der Engineer's Virtual Computer EVC1

Die EVC1-Karte wird von der Virtual Computer Corporation, USA, hergestellt (vgl. Abschnitt 3.3). Der Kern der von uns verwendeten Version ist ein Xilinx XC4013-5 FPGA. Die rekonfigurierbare Logik dieses FPGAs entspricht etwa 13.000 Gattern. Die Größe der realisierbaren Schaltungen ist also recht beschränkt. Als Taktsignal steht auf der Karte ein 40-MHz-Oszillator zur Verfügung. Alternativ kann das SBUS-Taktsignal verwendet werden, das bei unserer SPARCstation 10/40 eine Frequenz von 20 MHz hat. Ein frei wählbarer Takt für das FPGA ist nicht vorhanden.

Ein Teil der FPGA-Pins ist mit Steckern verdrahtet, auf denen eine Tochter-Platine mit zwei Megabyte statischem RAM (Zugriffszeit 35 Nanosekunden) sitzt. Die anderen Pins sind mit den SBUS-Signalen verknüpft. Wegen dieser Anordnung können Daten von der SPARCstation zum lokalen Speicher und zurück nur durch das FPGA übertragen werden. Eine direkte Übertragung, etwa durch einen DMA-Kanal, ist nicht möglich. Außerdem muß die Schnittstellen-Logik auf Seiten des EVC1 durch ein mitgeliefertes Slave-Interface-Makro im FPGA realisiert werden, und die EVC1-Karte kann nur durch speicher-orientierte Ein-/Ausgabe[1] angesprochen werden. Dies macht die Kommunikation recht langsam: Die Übertragung eines Wortes dauert etwa 500 Nanosekunden. Und die FPGA-Konfigurierung durch die SPARCstation benötigt sogar 200 Millisekunden. Wegen dieser Faktoren wäre die EVC1-Karte für kommerzielle Anwendungen nicht besonders geeignet. Als Plattform zur Demonstration unserer Konzepte ist sie jedoch ausreichend. Ein großer Vorteil für den Entwurf eines einfachen Prototypen sind die mitgelieferten, getesteten Software- und Hardware-Makros. Sie ermöglichen eine einfache Einbindung von Koprozessoren in ein auf der SPARCstation laufendes Programm, ohne daß die Schnittstelle oder die Hardware der Karte selbst angepaßt werden muß.

7.2 Der MOCKA-Compiler

Als Eingabesprache des MODULA PIPELINE COMPILER wurde MODULA-2 in erster Linie deshalb gewählt, weil für diese Sprache der MOCKA-Compiler[2] [Sch88a] im Quellcode zur Verfügung stand und größtenteils unverändert eingesetzt werden konnte. MOCKA ist ein relativ kleiner und übersichtlicher Übersetzer und schien deshalb für einen experimentellen Prototyp gut geeignet. Außerdem ist MODULA-2 selbst eine klare Sprache relativ kleinen Umfangs, die dennoch — wie bereits in Abschnitt 5.1.1 erläutert — alle wesentlichen Bestandteile einer imperativen Programmiersprache enthält. So mußten weniger für uns uninteressante Spracheigenschaften berücksichtigt werden. Und weil MODULA-2 keine schwierig zu beherrschenden, maschinennahen Konstrukte enthält, aber trotzdem durch die Unterbereichstypen eine Möglichkeit zur Spezifikation von beliebigen Bitbreiten bietet, konnten im Mo-

[1] engl. memory-mapped I/O
[2] MOCKA = MOdula Compiler KArlsruhe

DULA PIPELINE COMPILER Funktionen verwirklicht werden, die mit einer anderen Programmiersprache, etwa in einem C-Übersetzer, schwieriger oder gar nicht zu realisieren gewesen wären.

Im folgenden wird eine grobe Übersicht des Übersetzungsvorgangs im MOCKA gegeben. Darauf aufbauend werden dann die Ergänzungen im MODULA PIPELINE COMPILER vorgestellt. Von uns wurde nicht die ursprüngliche Version des MOCKA verwendet, sondern der PRISMA-Prototyp. Diese MOCKA-Erweiterung wurde ursprünglich zur automatischen Programm-Parallelisierung entwickelt [Lin92, Wei92, AW93]. Der PRISMA enthält leider keinen funktionsfähigen Optimierer, so daß auch die in Kapitel 8 vorgestellten Messungen ohne Übersetzer-Optimierungen durchgeführt werden mußten.

7.2.1 Phasen der Übersetzung

Der MOCKA ist ein Zwei-Paß-Übersetzer, dessen Funktionalität grob in die in Abbildung 7.1 gezeigten Phasen unterteilt werden kann:

Zunächst wird das Eingabeprogramm vom *Scanner* und vom *Parser* lexikalisch und syntaktisch analysiert und in die höhere Zwischensprache ASTA (Abstract Statements) übersetzt. Danach übersetzt der *Transformer* die abstrakte Syntax der Anweisungen in die niedrigere Zwischensprache MOBIL (Modula Backend Interface Language) [Sch88b]. Die semantische Analyse des Programms ist dabei auf den Parser und den Transformer verteilt. Schließlich erzeugt der *Code-Generator* aus den MOBIL-Instruktionen Objektcode, der vom *Binder* mit dem Code importierter Bibliotheken kombiniert und zu einem ausführbaren Programm gebunden wird.

MOBIL ist eine maschinennahe, aber von speziellen Prozessoreigenschaften unabhängige Zwischensprache. Das heißt, daß komplexe Berechnungen des MODULA-Programms auf der MOBIL-Ebene in elementare Befehle, die jeder Prozessor ausführen kann, aufgebrochen sind. Dies gilt jedoch nicht für die Adreßberechnungen in Feldzugriffen, die in MOBIL durch noch relativ komplexe Indizierungsfunktionen repräsentiert werden.

Für die PRISMA-Version des Übersetzers steht auch ein C-Backend zur Verfügung, mit dem C-Funktionen besonders einfach mit einem MODULA-Programm kombiniert werden können. Da dies für die Laufzeitfunktionen des EVC1 notwendig ist, wurde dieses Backend in dieser Arbeit verwendet.

7.3 Implementierung

Die Gesamtstruktur eines kombinierten Pipeline-Übersetzers wurde bereits in Abschnitt 6.2 (Abbildung 6.2) vorgestellt. Wir betrachten deshalb hier vor allem die Besonderheiten des MODULA PIPELINE COMPILER-Prototyps, insbesondere die Integration der Pipeline-Synthese in den MOCKA-Compiler und die automatische

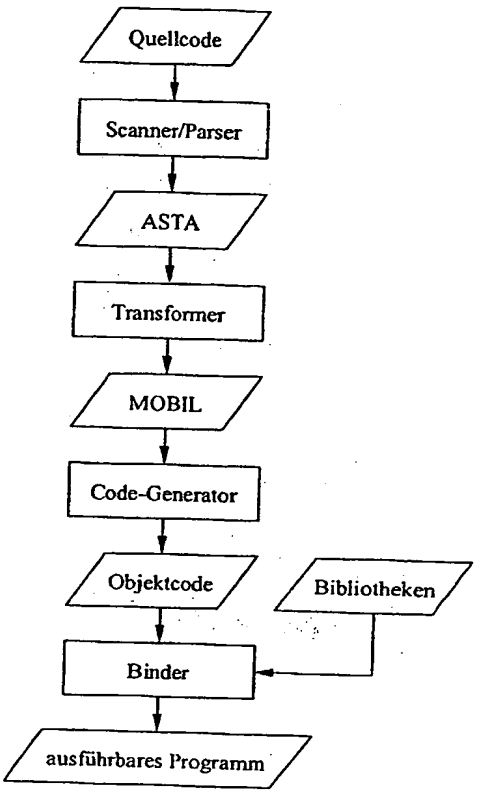


Abbildung 7.1: MOCKA-Übersetzungsphasen

Einbeziehung externer Entwurfswerkzeuge.

Beim Entwurf des Prototyps mußte zuerst entschieden werden, auf welcher Darstellung des Eingabeprogramms die Hardware-Synthese und die Partitionierung des Programms aufsetzt. Da viele der Transformationen aus Kapitel 5 und 6 auf der Ebene der Quellsprache eingeführt wurden, läge eine Transformation der höheren Zwischensprache ASTA nahe. Denn in dieser Darstellung sind die Strukturen des Quellprogramms noch nicht aufgelöst und lassen sich daher einfacher manipulieren. Andererseits sind aber die Programm-Instrumentierung und die Schätzung der Ausführungszeit einfacher in der maschinennahen Zwischensprache MOBIL durchzuführen. Denn die MOBIL-Instruktionen liegen im MOCKA auch als Kontrollfluß-Graph vor, und in einer Diplomarbeit im Rahmen des PRISMA-Projektes [Lin92] wurden bereits Erweiterungen vorgenommen, die in diesem Kontrollfluß-Graphen die ursprüngliche Struktur der Quellsprach-Konstrukte (Schleifen, bedingte Anweisun-

gen) rekonstruieren und eine Instrumentierung des MOBIL-Codes sehr erleichtern. Außerdem können die Indexausdrücke der Feldzugriffe bei der Vektorisierung auf relativ hoher Ebene analysiert werden, da sie bei der Übersetzung nach MOBIL erhalten bleiben. Da schließlich aus Aufwandsgründen für den Prototypen sowieso auf eine Implementierung der quellsprach-nahen Analysen und Transformationen (Algorithmen in den Abbildungen 5.19 und 5.20, Abschnitt 5.5) verzichtet werden mußte, wurde für den MODULA PIPELINE COMPILER ein auf der MOBIL-Ebene aufsetzender Entwurf gewählt.

Im folgenden beschreiben wir zunächst die Implementierung der Pipeline-Synthese und der Pipeline-Steuerungseinheit, bevor die Erzeugung einer FPGA-Konfiguration durch externe Entwurfswerkzeuge und die Integration der Koprozessor-Aufrufe in die Software erläutert werden.

7.3.1 Pipeline-Synthese

Wie oben erwähnt, wurde die automatische Kandidaten-Auswahl und -Normalisierung im Prototyp nicht implementiert. Statt dessen muß der Programmierer die Schleife, für die eine Pipeline synthetisiert werden soll, im Programmtext annotieren. Damit sichert er zu, daß die Schleife in der gewünschten Normalform ist.

Im Gegensatz zur Darstellung in Kapitel 5 generiert der MODULA PIPELINE COMPILER zuerst einen azyklischen Datenfluß-Graphen der annotierten Schleife. Danach wird das Einfügen der Rückkopplungszyklen mit der Vektorisierung kombiniert.

Azyklischer Datenfluß-Graph

Die Synthese des azyklischen Datenfluß-Graphen unterscheidet sich im MODULA PIPELINE COMPILER von dem in Abschnitt 5.1.4 vorgestellten Verfahren. Es werden keine separaten Graphen für jeden Basisblock aufgebaut und später durch eine Datenfluß-Kontrollfluß-Transformation verschmolzen. Vielmehr wird der ganze Schleifenrumpf, der nur noch aus Zuweisungen und bedingten Anweisungen besteht, in einem Durchlauf analysiert. Dieses Vorgehen entspricht der im "Transmogrifier C Compiler" (vgl. Abschnitt 3.2.1) verwendeten Methode. Jedoch wird der Datenfluß-Graph nicht auf Gatter-, sondern auf Operatorebene aufgebaut.

Beginnend mit der ersten Zuweisung des Schleifenrumpfes wird inkrementell für jeden Programmpunkt eine Liste der gültigen Werte aller bisher definierten Variablen erstellt. Unterschiedliche Vektor-Zugriffe werden dabei wie unabhängige Variablen behandelt. Die Werte der Variablen sind arithmetische oder logische Verknüpfungen der Eingabevariablen (oder -register). So entstehen Datenfluß-Graphen, in denen die Operatoren der MOBIL-Zwischensprache durch entsprechende Hardware-Operatoren ersetzt sind. An den "Verschmelzungspunkten" zweier Kontrollfluß-Pfade (nach bedingten Anweisungen) werden die korrekten Definitionen der betroffenen Variablen durch Multiplexer ausgewählt. Nach der Analyse aller Anweisungen ergibt die Zusammenfassung der Werte aller Ausgabevariablen den resultierenden

azyklischen Datenfluß-Graphen des ganzen Schleifenrumpfes. In diesem werden zuletzt die nach Definition 7 (Kapitel 5) äquivalenten Vektoreingaben durch Schieberegister verknüpft, so daß für sie nur ein Vektor-Port übrigbleibt.

Vektorisierung und Rückkopplungszyklen

Bei der Synthese des azyklischen Datenfluß-Graphen werden nebenbei alle Eingabe- und Ausgabe-Variablen, also die Mengen INPUT und OUTPUT aus Abschnitt 5.2.2, bestimmt. Deshalb kann jetzt für alle Eingabe-Register überprüft werden, ob sie von einer Ausgabe abhängen. In diesem Fall wird ein Rückkopplungszyklus in die Schaltung eingefügt. Jedoch müssen reguläre Vektor-Abhängigkeiten bereits in skalare Abhängigkeiten umgewandelt worden sein, da diese Transformation nicht implementiert ist. Wird bei der Analyse eine irreguläre Abhängigkeit festgestellt, ist die Schleife nicht vektorisierbar und die Pipeline-Synthese wird abgebrochen.

Pipelining

Zunächst wird aus der Verzögerung der Rückkopplungszyklen und der Anzahl der Vektor-Ports die erreichbare Pipeline-Zykluszeit T_C nach den Formeln (5.12) und (5.13) ermittelt. Für das Pipelining selbst wurde das ILP-basierte globale Optimierungsverfahren aus Abschnitt 5.4.2 jedoch noch nicht implementiert. Stattdessen werden die Register nach dem in Abschnitt 5.2.3 erwähnten einfachen Verfahren bei einer rekursiven Traversierung der Schaltung eingefügt. Um die Schaltungsgröße wenigstens etwas zu reduzieren, werden in einer anschließenden lokalen Optimierungsphase redundante Register zusammengefaßt.

Ausgabe als strukturelle VHDL-Datei

Schließlich wird die Pipeline-Schaltung in eine rein strukturelle Hardware-Beschreibung in VHDL umgesetzt und ausgegeben. Dabei wird für jeden Knoten der internen Repräsentation (Operator oder Register) eine VHDL-Komponente und für jede Kante ein entsprechendes VHDL-Signal erzeugt.

3.2 Pipeline-Steuerungseinheit

Für den MODULA PIPELINE COMPILER wurde eine einfache PCU-Version implementiert. Sie verfügt über drei Vektor-Ports für die Eingabe, aber nur über einen für die Ausgabe. Denn Schreibzugriffe auf den lokalen Speicher sind langsamer als Lesezugriffe, und die meisten Anwendungen benötigen mehrere Eingabe-, aber nur einen Ausgabe-Port. Abbildung 7.2 zeigt die EVC1-Karte mit dieser PCU.

Die Ports des Prototyps können auf Vektoren nur mit der Schrittweite eins zugreifen. Deshalb müssen andere Schrittweiten ausgeschlossen werden, und die Implemen-

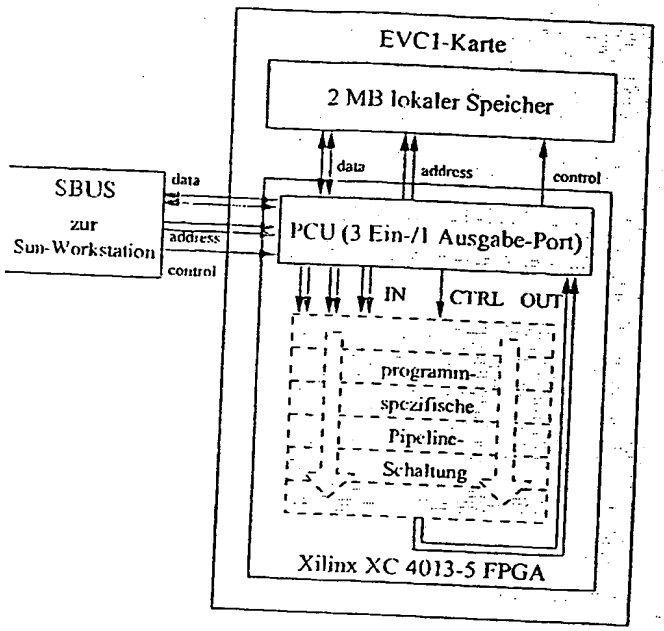


Abbildung 7.2: EVC1 mit prototypischer Pipeline-Steuerungseinheit

rung der Alias-Analyse und der Vektorisierung konnte entsprechend vereinfacht rden.

e PCU wurde so entworfen, daß sie mit dem SBUS-Takt (20 MHz) getaktet werden m. Für einen Lesezugriff auf den lokalen Speicher werden dann ein Takt und einen Schreibzugriff zwei Takte benötigt. Dies ergibt für die in Abschnitt 5.2.3 wendeten Parameter für die Zugriffszeiten die Werte $T_E = 50\ ns$ und $T_A = 100$ Wegen der vorhandenen Taktsignale und der internen Signalverzögerung der U-Komponenten konnte die Schnelligkeit der verwendeten RAM-Bausteine (35 leider nicht optimal ausgenutzt werden.

PCU wurde in VHDL spezifiziert. Mit den Entwurfswerkzeugen Synopsys FPGA mpiler [Syna, Synb] und Xilinx XACT [Xil94b, Xil94c] wurde daraus eine Netz- im XNF-Format, dem Standardformat zur Beschreibung einer Xilinx-FPGA- figuration, synthetisiert. Diese Netzliste enthält eine nicht instanzierte Kompo- e, die als Schnittstelle zur später integrierten programmspezifischen Pipeline t.

7.3.3 Generierung der FPGA-Konfiguration

Um eine funktionsfähige FPGA-Konfiguration zu erhalten, müssen die programmspezifischen Pipeline-Schaltungen mit der PCU verknüpft und in einen Konfigurations-Bitstrom transformiert werden. Dies geschieht automatisch durch die Kombination externer Entwurfswerkzeuge. Im Prototyp kann jedoch nur eine Pipeline pro Programm implementiert werden.

Zunächst wird die von der Pipeline-Synthese erzeugte VHDL-Datei in das XNF Format übersetzt, wobei die VHDL-Komponenten durch entsprechende FPGA-Makros für die Xilinx 4000-Serie ersetzt werden. Dies wird von dem MODULO-System zur bibliotheksorientierten Technologieabbildung [KKT+95] vorgenommen. MODULO optimiert die Schaltung nicht, ist dafür aber sehr schnell. Außerdem muß keine vollständige Operator-Bibliothek vorhanden sein. Denn die Ausnahmebehandlung von MODULO ruft automatisch ein mächtigeres Synthese-Werkzeug (z. B. Synopsys) auf, wenn ein Operator für eine bestimmte Wortbreite noch nicht vorhanden ist.

Danach werden die XNF-Netzlisten der PCU und der programmspezifischen Pipeline durch die Xilinx-XACT-Werkzeuge zusammengefügt, optimiert, plaziert und verdrahtet. Die resultierende Bitstrom-Datei wird schließlich von einem mit der EVC1-Karte gelieferten Hilfsprogramm in eine C-Header-Datei konvertiert, die in die Software integriert werden kann.

7.3.4 Anpassung des Zwischencodes

Wird der Parameter `-hw` verwendet, generiert der MODULA PIPELINE COMPILER eine FPGA-Konfiguration für eine annotierte Schleife. Wenn dies fehlerfrei gelingt, ruft das erzeugte Maschinenprogramm *immer* den Koprozessor auf, da im vorliegenden Prototyp keine automatische Partitionierung implementiert wurde. Der Aufruf geschieht durch die Instrumentierung des MOBIL-Codes mit Aufrufen von Laufzeit-Funktionen, die zum ausführbaren Code gebunden werden: Die Funktion `EVC_RUN` steuert die Pipeline-Ausführung und ersetzt im Programm die Instruktionen für die Software-Ausführung der Schleife. `EVC_RUN` erhält als Parameter die aktuelle Schleifenlänge und die Zahl der zum Füllen der Eingabe-Schieberegister nötigen Takte. Außerdem wird für alle Vektor-Ports die Adresse des ersten Wortes im lokalen Speicher (Basisadresse) sowie die Traversierungs-Richtung (für die Schrittweiten 1 und -1) angegeben. Dazu werden für die Felder in INPUT und OUTPUT Speicherbereiche im lokalen Speicher allokiert. Dabei wird auch überprüft, ob der lokale Speicher überhaupt groß genug ist. Falls in der Schleife auf Teilfelder zugegriffen wird, müssen außerdem Funktionen zur Berechnung der Position eines Teilfeldes innerhalb eines Feldes generiert werden.

Zusätzlich werden Funktionen zur Daten-Übertragung eingefügt. Dazu wird analysiert, in welchen umgebenden Schleifen des Kandidaten die Variablen aus INPUT und OUTPUT verwendet werden. Dann werden die Funktionen `EVC_PUT` und

EVC_GET in die jeweils äußerste Schleife eingefügt, für die dies der Fall ist. So finden zur Laufzeit so wenig Übertragungen wie möglich statt. Die Kopier-Funktionen für die Felder erhalten die oben berechneten Basisadressen im lokalen Speicher als Parameter.

Schließlich werden die Funktionen EVC_DOWNLOAD zur Konfigurierung und EVC_RESET zum Rücksetzen des EVCI eingefügt. Da nur ein Koprozessor pro Programm zugelassen ist, kann dies direkt am Programmanfang beziehungsweise am Programmende erfolgen.

7.3.5 Integration

Die in den vorhergehenden Abschnitten erläuterten Module wurden in den MOCKA-Compiler integriert. Abbildung 7.3 zeigt alle Übersetzungsphasen im MODULA PIPELINE COMPILER, wobei neue Programmteile des Übersetzers sowie Hardware- und Software-Komponenten des Laufzeitsystems grau unterlegt dargestellt sind. Der MOCKA selbst wird folgendermaßen erweitert: Nach der Transformation in MOBIL-Code wird die Pipeline-Synthese durchgeführt. Sie erzeugt die VHDL-Datei `pipe.vhdl`. Danach wird der MOBIL-Code instrumentiert, bevor er vom Code-Generator weiterverarbeitet wird.

Außerdem ruft der Übersetzer einen Shell-Skript auf, der die externen Entwurfswerkzeuge steuert. Diese synthetisieren aus `pipe.vhdl` zusammen mit der PCU-Netzliste `pcu.xnf` die FPGA-Konfiguration und speichern sie als C-Header-Datei `pipe.h`. Danach übersetzt ein C-Übersetzer die Laufzeit-Funktion EVC_DOWNLOAD (in `rt_downl.c`) zusammen mit `pipe.h`. Der erzeugte Objektcode in `rt_downl.o` konfiguriert die EVCI-Karte mit der programmspezifischen Pipeline. `rt_downl.o` wird dann — zusammen mit den vorübersetzten, nicht veränderlichen Funktionen der Bibliothek (EVC_PUT, EVC_GET, EVC_RUN und EVC_RESET) — zu dem instrumentierten MODULA-Programm gebunden.

Da die Hardware-Synthese unter Umständen mehrere Stunden dauern kann, wird die verwendete Pipeline-Beschreibung `pipe.vhdl` nach jeder erfolgreichen Ausführung des Shell-Skripts umbenannt und gespeichert, so daß beim nächsten Übersetzungs-Lauf eine Änderung in der Pipeline festgestellt werden kann und die Synthese nur in diesem Fall ausgeführt werden muß.

7.4 Erfahrungen und Verbesserungsvorschläge

Die Arbeit mit dem MODULA PIPELINE COMPILER zeigte, daß sich MODULA-2 zur Spezifikation von SP-Rechner-Anwendungen recht gut eignet. Es gab keine Beispiele, die aufgrund irgendwelcher Besonderheiten dieser Sprache nicht implementiert werden konnten. Ebenso hat die Entscheidung, die Pipeline-Synthese auf der MOBIL-Ebene aufzusetzen, zu keinen unerwarteten Schwierigkeiten geführt.

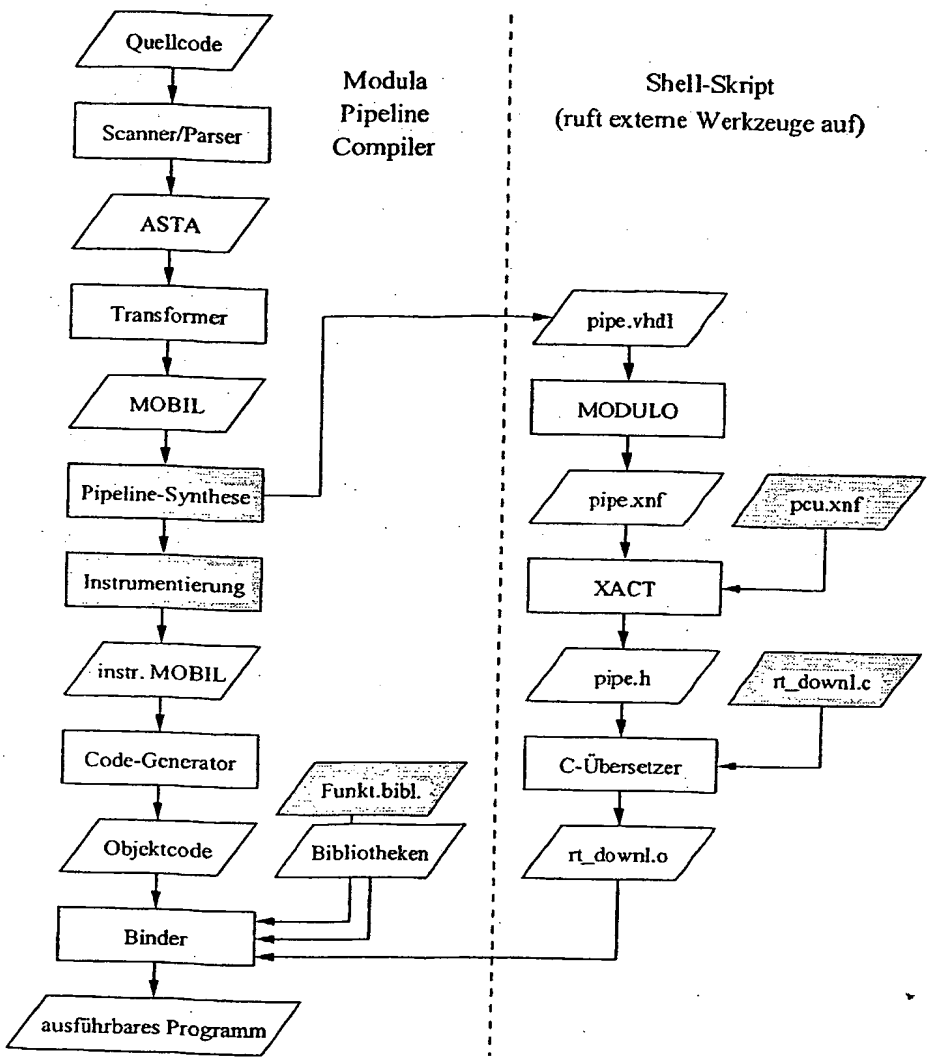


Abbildung 7.3: MPC-Übersetzungsphasen

ich der auf drei Vektor-Eingabe-Ports und einen Vektor-Ausgabe-Port mit der tgelegten Schrittweite eins beschränkte PCU-Prototyp war für alle betrachte- i Beispiele ausreichend. Offensichtlich kommen kompliziertere Zugriffsstrukturen i den für SP-Rechner geeigneten Anwendungen selten vor. Jedoch verhinderte r nur sequentiell mögliche Zugriff auf den lokalen Speicher für einige Programme e Beschleunigung durch die rekonfigurierbare Hardware. Dieser Engpaß kann je- ch nur durch die in Abschnitt 9.4 diskutierte Verwendung einer RE mit mehreren eicherbänken behoben werden.

llerdings stellte sich heraus, daß die Implementierung der Schnittstelle zwischen irt und RE für viele Anwendungen unzureichend ist: Einerseits werden beim Ko- eren von Feldern auf die EVC1-Karte immer 32 Bit übertragen, was z. B. bei yte-Feldern zur Verschwendung von 24 Bit pro Wort, also zu einer sehr schlechten utzung der Übertragungsbandbreite, führt. Folglich ergeben sich sehr lange Kom- unikationszeiten, wie man auch an den im nächsten Kapitel vorgestellten Meßer- ebnissen erkennt. Andererseits können mit dem PCU-Prototyp nur wenige skalare ariablen übertragen werden, weshalb für manche Anwendungen gar kein Hardware- oprozessor synthetisiert werden kann. Wir geben deshalb im nächsten Abschnitt nige Hinweise auf mögliche Verbesserungen der Schnittstellen-Implementierung.

.4.1 Schnittstellen-Optimierungen

unächst könnte die Übertragungsbandbreite der Schnittstelle besser genutzt wer- en, wenn elementare Daten mit kurzen Wortlängen oder gar einzelne Bits zu ei- em ganzen Wort zusammengefaßt und gepackt übertragen würden. Dazu wären usätzliche Anweisungen auf dem Wirt sowie parametrisierbare Parallel/Seriell- nd Seriell/Parallel-Wandler in der PCU nötig, um die Daten vor dem Kopie- en zu packen und danach wieder zu entpacken. Der Übersetzer müßte außerdem bschätzen, ob sich der hardware- und software-seitige Aufwand hierfür lohnt.

Auf dieselbe Weise könnte auch die Übertragung vom lokalen Speicher in das FPGA ptimiert werden, wodurch auch dieser Speicher besser genutzt und der Pipeline- Durchsatz weniger eingeschränkt würde. Bei kleinen, nur wenige Bit breiten Opera- oren ließe sich dann ein sehr hoher Durchsatz erreichen. Jedoch müßte eine erwei- erte Steuerung die Pipeline anhalten, wenn Speicherzugriffe notwendig werden.[3]

Weitergehend können Pipelines, in denen nur ein Vektor-Port gelesen oder geschrie- ben wird, durch Überlappung der Pipeline-Ausführung mit der Vektor-Übertragung vom beziehungsweise zum Wirt beschleunigt werden.[4] Jedoch sind dann weitere Schleifen-Transformationen nötig, um bei mehreren Schleifen-Ausführungen die er- ste beziehungsweise letzte Ausführung optimieren zu können.

[3]Es müßte also eine Ausnahmebehandlung implementiert werden. Ein ähnliches Problem wurde bei der in Anhang A vorgestellten Schaltung gelöst.

[4]Dies wurde in der PCU für die manuell durchgeführten Experimente in [Wei96c] bereits implementiert.

Auch die Eingabe und Ausgabe von skalaren Werten ist bisher wenig zufriedenstellend gelöst, indem die skalaren Ports für die Initialisierung und das Auslesen der Ergebnisse mit allen Registern verdrahtet sind. Wegen des hohen Verdrahtungs- und Logik-Aufwands ist dieses Verfahren nur für wenige Eingaben und Ausgaben brauchbar. Um mehr Register ein- und auslesen zu können, könnten sie beispielsweise wie die Vektor-Eingabe-Register zu einem von der PCU gesteuerten, nur in der Initialisierungs- und Auslesephase aktiven Schieberegister verknüpft werden. Oder es könnten neuere, speziell für Koprozessor-Anwendungen entwickelte FPGA-Familien verwendet werden, bei denen das Problem des Zugriffs auf interne Register bereits bei der Entwicklung des FPGAs berücksichtigt wurde: So kann etwa bei der XC6200-Serie von Xilinx ein spezielles Interface schnell auf jedes interne Register zugreifen, so daß dafür keine Vorsorge getroffen werden muß.

Kapitel 8

Anwendungen und Ergebnisse

In diesem Kapitel werden einige Anwendungsbeispiele vorgestellt, die mit dem MODULA PIPELINE COMPILER übersetzt und beschleunigt werden können. Bei allen Beispielen wurde die Laufzeit der betroffenen Programmteile mit und ohne Beschleunigung durch die EVC1-Karte gemessen. Neben der Gesamtdauer der Koprozessor-Aufrufe (einschließlich Kommunikation und Konfigurierung) wurde auch die reine Pipeline-Ausführungszeit gemessen. So kann die "asymptotisch erreichbare Beschleunigung" (bei vollständiger Elimination der Zusatzkosten) berechnet werden.

Neben den hier erwähnten Programmklassen sind die in den Kapiteln 5 und 6 vorgestellten Verfahren prinzipiell auf viele weitere Klassen anwendbar. Wegen der Einschränkungen des Prototyps können jedoch nicht alle mit dem MODULA PIPELINE COMPILER verarbeitet werden. Außerdem wurde auf die Aufführung von Beispielen verzichtet, die nur "asymptotisch beschleunigt" werden, also tatsächlich mit der EVC1-Karte langsamer als in Software laufen. Gleichwohl können solche Programme auf anderen SP-Rechnern unter günstigeren Rahmenbedingungen beschleunigt werden.

Die Messungen der Pipeline-Ausführungs-, Kommunikations- und Konfigurierungszeiten sind eigentlich nicht wirklich interessant. Denn sie bestätigen die Vorhersagen, die sich aus der Schleifenlänge, der Zykluszeit und der Größe der Felder ergeben, recht genau. Interessant ist vielmehr die gemessene Software-Laufzeit und deren Verhältnis zur Hardware-Laufzeit. Denn genaue Methoden zur Software-Schätzung konnten im Rahmen dieser Arbeit nicht mehr untersucht werden. Sie sind auch aus prinzipiellen Gründen schwieriger, wie wir bereits in Abschnitt 6.4.1 erläutert haben. Ein direkter Software/Hardware-Vergleich kann also nur durch Messungen erfolgen.

Im nächsten Abschnitt wird zunächst der verwendete Meßaufbau erläutert, bevor die einzelnen Beispiele vorgestellt werden. Der letzte Abschnitt dieses Kapitels zieht Folgerungen aus den Ergebnissen und extrapoliert sie für eine Zielplattform, die dem technischen Stand von 1997 entspricht.

8.1 Der Meßaufbau

Für die Messungen wurde der MODULA PIPELINE COMPILER auf der in Abschnitt 7.1 vorgestellten Hardware-Plattform verwendet. Dabei konnten Programmlaufzeiten nur mit UNIX-Systemaufrufen gemessen werden. Diese haben eine Genauigkeit von einer Millisekunde, was bei der Messung von größeren Programmregionen ausreichend ist.

Die Messungen geben immer die Zeitdifferenz (elapsed time) zwischen zwei Programmpunkten, also zwischen dem Beginn und dem Ende der untersuchten Region, an. Dies ist die Zeit, die ein Anwender warten muß, bis das Ergebnis der Berechnung vorliegt. Es wäre falsch, die reine Prozessorzeit zu messen, da dies nur für die Software korrekt ist. Denn die EVC1-Karte kann auch Berechnungen in Hardware durchführen, solange der Prozessor einen anderen Prozeß bearbeitet. Diese Zeit würde also nicht als Rechenzeit des Prozesses gemessen werden und die Messung zugunsten der Hardware verfälschen. Andererseits wird aber die als Zeitdifferenz gemessene Ausführungszeit eines Prozesses durch Prozeßwechsel verlängert, was sich wiederum stärker auf reine Software-Prozesse auswirkt. Andere Prozesse (Betriebssystem, Netzwerk) lassen sich jedoch in Multiuser-Betriebssystemen nicht ganz ausschließen. Deshalb wurden die Messungen mit einem möglichst unbelasteten Rechner durchgeführt, was zu wiederholbaren Ergebnissen führte. Die angegebenen Werte sind der Mittelwert von jeweils 100 Meßdurchläufen.

Die Laufzeiten wurden mit Programmen gemessen, die ohne Übersetzer-Optimierungen entstanden, da der Prototyp des MODULA PIPELINE COMPILER keine Standard-Optimierungen enthält. Leider konnten auch keine Untersuchungen über den Einfluß unterschiedlicher Cachegrößen oder ähnlicher Systemparameter sowie über deren Auswirkungen auf Übersetzer-Optimierungen durchgeführt werden.

8.2 Beispiele

In diesem Abschnitt werden Beispielprogramme aus den Bereichen FIR-Filter, Zellularautomaten, Bildverarbeitung sowie Mustererkennung in Zeichenketten vorgestellt. Für jedes der Programme werden Messungen der Software- und Hardware-Laufzeit, der erreichten Beschleunigungen sowie des Hardware-Bedarfs der Koprozessoren (einschließlich PCU) präsentiert.

8.2.1 FIR-Filter

Die erste betrachtete Programmklasse sind FIR-Filter[1]. Sie werden in der digitalen Signalverarbeitung für viele Zwecke eingesetzt. Wir betrachten hier ein achtstufiges Filter zur Signalglättung, das jeden Wert durch den Durchschnitt der acht umge-

[1]FIR = Finite Impulse Response

```
CONST N = 200000;
VAR   X:       ARRAY [0..N-1] OF CARDINAL;
      I, ITER: INTEGER;
...
FOR ITER := 1 TO MAXITER DO
  FOR I := 0 TO N-8 DO
    X[I] := X[I] DIV 8;
    X[I] := X[I] + X[I+1] DIV 8;
    X[I] := X[I] + X[I+2] DIV 8;
    X[I] := X[I] + X[I+3] DIV 8;
    X[I] := X[I] + X[I+4] DIV 8;
    X[I] := X[I] + X[I+5] DIV 8;
    X[I] := X[I] + X[I+6] DIV 8;
    X[I] := X[I] + X[I+7] DIV 8;
  END;
END;
```

Abbildung 8.1: Einfaches FIR-Filter

benden Werte ersetzt. Da nur konstante Koeffizienten mit dem Wert $\frac{1}{8}$ auftreten, kann dieses Filter in nur einem FPGA implementiert werden. Abbildung 8.1 zeigt das einfache Filterprogramm. Es führt dieselbe Funktion wie unser Programmbeispiel aus Abbildung 5.1 aus, operiert aber nur auf *einem* Vektor der Länge 200000. Die veränderten Werte werden also direkt in denselben Vektor zurückgeschrieben. Die resultierende Pipeline entspricht der Beispiel-Pipeline aus Kapitel 5 (siehe Abbildung 5.8) mit acht Filterstufen. Da sie nur einen Vektor-Eingabe- und einen Vektor-Ausgabe-Port benützt, beträgt der Pipeline-Taktzyklus 150 ns.

Algorithmus	einfaches FIR-Filter				kombiniertes FIR-Filter			
Iterationszahl	1		10		1		10	
Dauer/Beschleunig.	D [s]	B	D [s]	B	D [s]	B	D [s]	B
Software-Laufzeit	1.020		10.276		1.704		15.497	
Pipeline-Ausführung	0.030	34.00	0.300	34.25	0.030	56.80	0.300	51.66
HW m. Komm./Konf.	0.455	2.24	0.706	14.56	0.611	2.79	0.914	16.96

Tabelle 8.1: Laufzeit-Messungen FIR-Filter

Tabelle 8.1 zeigt die gemessenen Laufzeiten dieses Programms und des erweiterten Programmbeispiels aus Abbildung 5.9, das einen Zufallszahlengenerator mit einem FIR-Filter (für die Messungen ebenfalls achtstufig und mit Vektorlänge 200000) kombiniert. Die reine Ausführungsdauer der Pipeline ist bei beiden Programmen gleich, da sie nur von der Schleifenlänge und dem Pipelinetakt abhängt. Dagegen ist die gesamte Hardware-Ausführung (letzte Zeile in Tabelle 8.1) für das kombinierte FIR-Filter länger, da dieses Programm die Eingaben und Ausgaben des Filters in getrennten Vektoren speichert, die beide zum EVC1 und zurück kopiert

werden müssen. Da jedoch das kombinierte Filter mehr Operationen in einem Zyklus durchführt, ist auch seine Software-Laufzeit länger, so daß sich insgesamt eine höhere Beschleunigung als bei dem einfachen FIR-Filter ergibt.

Bei beiden Beispielen belegt der Koprozessor 43% der Hardware-Ressourcen des verwendeten XC4013 FPGA.

8.2.2 Zellularautomaten

Zellularautomaten — bzw. genaugenommen Programme zur Simulation von Zellularautomaten — stellen eine weitere geeignete Anwendungsklasse für den MODULA PIPELINE COMPILER dar, da auch hier gleichartige Operationen auf großen Datenmengen durchgeführt werden. Außerdem kann bei den meisten Automaten der Zustand jeder Zelle mit wenigen Bit repräsentiert werden, und die Zustandsübergangsfunktion entspricht einem endlichen Automaten.

Als Beispiel eines Zellularautomaten verwenden wir das bekannte "Game of Life": In dem Spiel haben Zellen in einer zweidimensionalen Ebene entweder den Zustand *lebendig* oder *tot*. Die Zellen ändern ihren Zustand gleichzeitig einmal pro *Generation*. Der neue Zustand hängt von dem alten Zustand der Zelle selbst und ihrer acht direkten Nachbarn folgendermaßen ab:

- Eine lebendige Zelle überlebt, wenn zwei oder drei Nachbarn lebendig sind.
- Eine tote Zelle wird lebendig, wenn genau drei Nachbarn lebendig sind.
- Andernfalls ist die Zelle in der nächsten Generation tot.

Das Programm in Abbildung 8.2 implementiert dieses Spiel. Aus seiner inneren Schleife wird der in Abbildung 8.3 dargestellte Datenfluß-Graph erzeugt.

Mit diesem Programm wurde die erste und die hundertste nachfolgende Generation einer gegebenen 400x400 Zellkonfiguration berechnet. Tabelle 8.2 zeigt die gemessenen Software- und Hardware-Laufzeiten und die erreichten Beschleunigungen.

berechnete Generationen	1		100	
Dauer/Beschleunigung	D [s]	B	D [s]	B
Software-Laufzeit	1.658		173.420	
Pipeline-Ausführung	0.044	37.68	4.761	36.43
HW mit Komm./Konf.	0.608	2.73	5.326	32.56

Tabelle 8.2: Laufzeit-Messungen "Game of Life"

Besonders bei der Berechnung vieler aufeinanderfolgender Generationen ergibt sich eine hohe Beschleunigung. Denn die Hardware kann in jedem Pipeline-Takt den

```
TYPE  PLAIN     = ARRAY[0..VERLEN-1],[0..HORLEN-1] OF [0..1];
      TWOPLAINS = ARRAY[0..1] OF PLAIN;
VAR   P:                     TWOPLAINS;
      ITER, V, H, SRC, DEST: INTEGER;
      SUM:                   [0..15];
...
SRC  := 0;
DEST := 1;
FOR ITER := 1 TO MAXITER DO
  (* Compute next generation. Result goes in P[DEST]. *)
  FOR V := 1 TO VERLEN-2 DO
    FOR H := 0 TO HORLEN-3 DO
      SUM := (P[SRC,V-1,H  ] + P[SRC,V-1,H+1] + P[SRC,V-1,H+2]) +
             (P[SRC,V  ,H  ] +                  P[SRC,V  ,H+2]) +
             (P[SRC,V+1,H  ] + P[SRC,V+1,H+1] + P[SRC,V+1,H+2]);
      IF (SUM = 3) OR ((SUM = 2) AND (P[SRC,V,H+1] = 1)) THEN
        P[DEST,V,H+1] := 1;
      ELSE
        P[DEST,V,H+1] := 0;
      END;
    END;
  END;
  (* Switch source and result. *)
  DEST := SRC;
  SRC  := 1 - SRC;
END;
```

Abbildung 8.2: Game of Life

Folgezustand einer Zelle berechnen. Diese logische Verknüpfung der Zustände ist in Software nur mit relativ vielen Additions- und Vergleichs-Befehlen möglich. Der Pipeline-Durchsatz könnte durch eine Optimierung des Zugriffs auf den lokalen Speicher sogar noch weiter erhöht werden (vgl. Abschnitt 7.4.1). Denn in der vorliegenden Implementierung wird von jedem 32-Bit-Wort im Speicher nur ein Bit verwendet, die E/A-Bandbreite also nur sehr schlecht ausgenützt.

Ebenso wird über die Schnittstelle zwischen Wirt und RE für jede Zelle ein ganzes Wort übertragen, obwohl ein Bit ausreichen würde. Folglich kann auch die Kommunikationszeit signifikant reduziert werden. Aus diesen Gründen ist für diese Anwendung – auch unter Beibehaltung der EVCI-Karte als Ziel-Hardware — noch eine erhebliche Leistungssteigerung gegenüber den hier gemessenen Werten möglich. Eine kompliziertere PCU wäre auch im XC4013 noch implementierbar, da der Koprozessor mit der momentan verwendeten PCU nur 26% der FPGA-Ressourcen belegt.

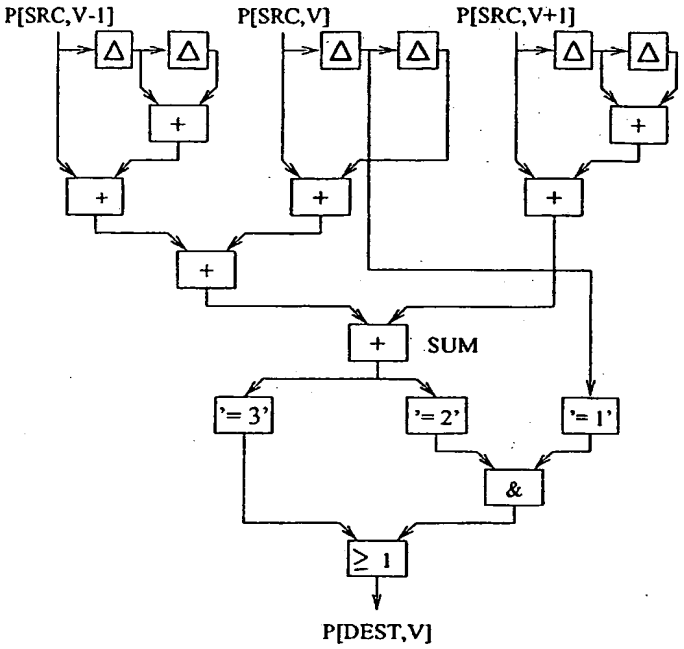


Abbildung 8.3: Datenflußgraph "Game of Life"

8.2.3 Bildverarbeitung

Hier stellen wir die genauen Laufzeit-Messungen der in Kapitel 2 ausführlich betrachteten Bildverarbeitungsalgorithmen vor. Sie sind den vorgenannten Beispielen insofern ähnlich, als sie auch als Zellularautomaten oder als zweidimensionale Filter betrachtet werden können. Der Bildglättungs-Koprozessor belegt 30% und der Kantendetektions-Koprozessor 41% der FPGA-Ressourcen.

Algorithmus	Bildglättung				Kantendetektion			
Iterationszahl	1		10		1		10	
Dauer/Beschleunigung	D [s]	B	D [s]	B	D [s]	B	D [s]	B
Software-Laufzeit	0.563		5.632		0.906		8.810	
Pipeline-Ausführung	0.026	21.65	0.263	21.43	0.027	33.81	0.266	33.15
HW mit Komm./Konf.	0.439	1.28	0.690	8.17	0.437	2.07	0.681	12.94

Tabelle 8.3: Laufzeit-Messungen Bildverarbeitung

3.2.4 Verarbeitung von Zeichenketten

Schließlich eignen sich viele Algorithmen, die Zeichenketten verarbeiten, für eine Implementierung auf SP-Rechnern, da auch hier iterative Operationen auf Daten kurzer Wortlänge (in der Regel acht Bit) ausgeführt werden. Wir demonstrieren dies an einem einfachen Programm (Abbildung 8.4) zur exakten Mustererkennung: Der Ausgabevektor `Y` enthält an denjenigen Positionen eine eins, an denen in dem Eingabetext `X` (Länge `N`) eine vorgegebene, kürzere Zeichenkette `P` (das *Muster* der Länge `L`) beginnt. Abbildung 8.5 zeigt den generierten Datenflußgraphen.

```
VAR X    : ARRAY[0..N-1] OF CHAR;
    Y    : ARRAY[0..N-1] OF [0..1];
    I, J : CARDINAL;
...
FOR I := 0 TO N-L DO
  Y[I] := 1;
  FOR J := 0 TO L DO
    IF X[I+J] <> P[J] THEN
      Y[I] := 0;
    END;
  END;
END;
```

Abbildung 8.4: Mustererkennung in Zeichenketten

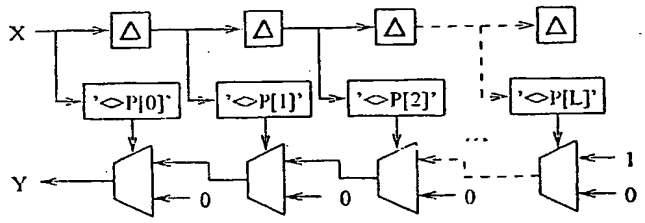


Abbildung 8.5: Datenflußgraph Mustererkennung

Die Laufzeitmessungen in Tabelle 8.4 wurden für einen Eingabetext der Länge 100000 vorgenommen. Es wurden zwei verschiedene Programme — mit Mustern der Länge 10 und 20 — gemessen. Hier ergibt sich die Beschleunigung, da jede Position des Eingabetextes in einem Pipeline-Takt mit dem ganzen Muster verglichen wird. Deshalb ist die Dauer der Pipeline-Ausführung auch unabhängig von der Länge des Musters, während die Software-Laufzeit natürlich direkt von ihr abhängt.

Auch für dieses Programm gilt die im vorhergehenden Abschnitt gemachte Bemerkung, daß das Ergebnis durch Schnittstellen-Optimierungen erheblich verbessert

werden könnte. Da die Koprozessoren nur 26% bzw. 28% der Hardware-Ressourcen belegen, wäre dies problemlos auf dem EVC1 möglich. Auch könnten noch wesentlich längere Muster verarbeitet werden, was zu entsprechend skalierten Beschleunigungen führen würde.

Länge des Musters	10		20	
Dauer/Beschleunigung	D [s]	B	D [s]	B
Software-Laufzeit	0.767		1.443	
Pipeline-Ausführung	0.015	51.13	0.015	96.20
HW mit Komm./Konf.	0.382	2.00	0.384	3.76

Tabelle 8.4: Laufzeit-Messungen Mustererkennung

Das in Abschnitt 3.4 erwähnte Problem, die Editierdistanz zweier ähnlicher, aber nicht exakt gleicher Zeichenketten zu bestimmen, kann mit der vorliegenden Version des MODULA PIPELINE COMPILER nicht bearbeitet werden. Dazu wäre eine Erweiterung der Schnittstelle zum Lesen vieler skalarer Ausgabewerte nötig, wie sie in Abschnitt 7.4.1 vorgeschlagen wurde.

8.3 Folgerungen

Die Messungen zeigen, daß sich für die untersuchten Beispiele durchweg eine reine Beschleunigung (ohne Berücksichtigung der Kommunikations- und Konfigurierungszeiten) von einer bis zwei Größenordnungen ergibt. Dieser Faktor wäre bei Koprozessoren, die die Ressourcen des XC4013 stärker auslasten, also mehr Operationen parallel durchführen, noch größer. Denn die Auslastung liegt bei allen betrachteten Beispielen unter 50%, zum Teil sogar unter 30%. Noch höhere Beschleunigungsfaktoren können erreicht werden, wenn unsere weitgehend maschinenunabhängigen Übersetzungsmethoden für größere SP-Rechner mit mehreren FPGAs eingesetzt werden und tiefe Pipelines mit hohem Parallelitätsgrad synthetisiert werden.

Wegen der bereits mehrfach erwähnten Unzulänglichkeiten der EVC1-Karte sind die erreichten Gesamtbeschleunigungen (einschließlich Zusatzkosten) jedoch leider etwas unbefriedigend. Die EVC1-Karte entspricht aber etwa dem technischen Stand von 1993. Für neuere SP-Rechner können unter günstigeren Rahmenbedingungen wesentlich bessere Meßwerte, die näher an den asymptotischen Grenzwerten liegen, erwartet werden.

Dies soll mit einer Modellrechnung, die realistische Annahmen für eine FPGA-Karte auf dem technischen Stand von 1997 trifft, verdeutlicht werden: Wir gehen von einem PCI-Bus-System aus, bei dem mit DMA-Kanälen eine Übertragungsrate von etwa 100 MB/s erreicht werden kann. Dies entspricht bei 32-Bit-Worten einer Kommunikationsdauer von circa 40 Nanosekunden pro Wort, ist also etwa 12 mal schneller

als die beim EVC1 verwendete Implementierung der SBUS-Schnittstelle (500 Nanosekunden pro Wort). Als rekonfigurierbare Hardware wird ein neues XC6216 FPGA (vgl. Abschnitt 3.1.2) verwendet, dessen Gatterkomplexität etwa dem im EVC1 verwendeten XC4013 entspricht, das aber wesentlich schneller konfiguriert werden kann: Wir nehmen eine Konfigurierungszeit von einer Millisekunde über den PCI-Bus an, also eine etwa 200 mal schnellere Konfigurierung als beim EVC1. Schließlich wird bei Verwendung von SRAM-Bausteinen mit 20 Nanosekunden Zugriffszeit eine doppelt so hohe Taktfrequenz (circa 40 MHz) für lokale Speicherzugriffe möglich, wodurch sich die erreichbare Pipeline-Taktfrequenz auch verdoppelt.

Architektur	EVC1/SBUS			XC6216/PCI-Bus		
Ausführungszeiten	T_P [s]	T_{HW} [s]	T_P/T_{HW}	T_P [s]	T_{HW} [s]	T_P/T_{HW}
einf. FIR-Filter (1 x)	0.030	0.455	6.6%	0.015	0.034	44.1%
einf. FIR-Filter (10 x)	0.300	0.706	42.5%	0.150	0.167	89.8%
komb. FIR-Filter (1 x)	0.030	0.611	4.9%	0.015	0.046	32.6%
komb. FIR-Filter (10 x)	0.300	0.914	32.8%	0.150	0.184	81.5%
Game of Life (1 x)	0.044	0.608	7.2%	0.022	0.052	42.3%
Game of Life (100 x)	4.761	5.326	89.4%	2.381	2.411	98.9%
Bildglättung (1 x)	0.026	0.439	5.9%	0.013	0.031	41.9%
Bildglättung (10 x)	0.263	0.690	38.1%	0.132	0.151	87.4%
Kantendetektion (1 x)	0.027	0.437	6.2%	0.014	0.032	43.8%
Kantendetektion (10 x)	0.266	0.681	39.1%	0.133	0.151	88.1%
Mustererkennung	0.015	0.382	3.9%	0.008	0.022	36.4%

Tabelle 8.5: Architekturvergleich der Hardware-Ausführungszeiten

Die Hardware-Ausführungszeiten aller in diesem Kapitel aufgeführten Beispielprogramme wurden auf diese XC6216/PCI-Bus-Architektur übertragen. Tabelle 8.5 stellt die auf der EVC1/SBUS-Architektur gemessenen Ausführungszeiten für die Pipeline (T_P) und für den ganzen Hardware-Aufruf (T_{HW}) den Voraussagen für diese neue Architektur gegenüber. Dabei wird für jede Architektur auch der prozentuale Anteil der reinen Pipeline-Ausführungszeit an der Gesamtzeit der Hardware-Ausführung (T_P/T_{HW}) angegeben.

Die Tabelle zeigt folgendes: Die reinen Pipeline-Ausführungszeiten werden durch den schnelleren Speicher zwar halbiert, aber wegen der auch in der neuen Architektur recht beschränkten Hardware-Ressourcen (nur ein FPGA) nicht weiter reduziert. Jedoch sind die Kommunikations- und Konfigurierungszeiten bei modernen Bus- und FPGA-Architekturen wesentlich geringer als beim EVC1, so daß das beim EVC1 gemessene krasse Mißverhältnis zwischen tatsächlicher Berechnungszeit und Zusatzkosten (z. T. weniger als 5% für die eigentliche Berechnung) nicht mehr auftritt: Bei der modellierten Architektur beträgt der Berechnungs-Anteil immerhin zwischen circa 35% und 99%. Die neueren Entwicklungen kommen also den Anforderungen an

die Geschwindigkeit der Kommunikation und der Konfigurierung, die sich in dieser Arbeit herausgestellt haben, entgegen, so daß für zukünftige Architekturen mit einer weiter verbesserten Eignung für pipeline-basierte Koprozessoren gerechnet werden kann. Jedoch darf gleichzeitig nicht vergessen werden, daß auch neuere Mikroprozessoren wesentlich schneller als der von uns verwendete SuperSPARC-Prozessor sind.

Insgesamt haben die Ergebnisse der mit dem MODULA PIPELINE COMPILER durchgeführten Experimente gezeigt, daß geeignete Programme mit den in dieser Arbeit entwickelten Verfahren durch rekonfigurierbare Koprozessoren signifikant beschleunigt werden können, obwohl die Ergebnisse durch die Verwendung der inzwischen veralteten EVC1-Karte ungünstiger als erhofft ausfielen.

Kapitel 9

Erweiterungen

Dieses Kapitel zeigt verschiedenartige Erweiterungsmöglichkeiten der pipelinebasierten Strukturprogrammierung auf. Dies betrifft einerseits Erweiterungen der Analyse- und Syntheseverfahren unter Beibehaltung des in Kapitel 6 zugrundegelegten Maschinenmodells. Andererseits werden auch Änderungen und Erweiterungen betrachtet, die für komplexere strukturprogrammierbare Rechner benötigt werden, und mögliche Erweiterungen der Eingabesprache diskutiert.

9.1 Synthese-Erweiterungen

Um für mehr Programme lohnende Pipelines synthetisieren zu können, sind vor allem die beiden im folgenden genannten Erweiterungen relevant. Daneben sind weitere Schaltungs-Optimierungen möglich, die sich jedoch nicht so stark auf die Leistungsfähigkeit der Pipelines auswirken.

9.1.1 Spezielle kombinatorische Funktionen

In Abschnitt 5.1 wurde bereits erwähnt, daß die Vektorisierung von in Schleifen auftretenden speziellen kombinatorischen Funktionen ein besonderes Beschleunigungspotential bietet. Denn wenn diese Funktionen, die schon isoliert wesentlich schneller auf einer RE als auf dem Wirt berechnet werden können, iterativ und parallel in einer Pipeline ausgeführt werden, wird die Berechnung nochmals um den Beschleunigungsfaktor der Pipeline schneller.

Eine einfache Integrationsmöglichkeit stellt die Annotation dieser Funktionen im Eingabeprogramm dar, wenn für sie ein (rein kombinatorischer) benutzerdefinierter Hardware-Operator zur Verfügung gestellt wird. Die Funktionen können dann von der Pipeline-Synthese wie gewöhnliche Operatoren behandelt werden.

Alternativ können die Operatoren auch automatisch (wie im PRISM-System durch Logiksynthese) aus dem Eingabeprogramm generiert werden. Dazu sind jedoch ge-

naue Kriterien zur automatischen Erkennung dieser Funktionen nötig, die noch entwickelt werden müssen. Sind diese Kriterien erfüllt, werden die Anweisungen einer Funktion zur Synthese eines speziellen kombinatorischen Operators zusammengefaßt. Andernfalls bleiben die Anweisungen unverändert und werden einzeln vektorisiert.

Ein manueller Schaltungsentwurf, in dem die Integration solcher Funktionen zu einer sehr großen Beschleunigung führte, wird in Anhang A vorgestellt. Dabei wird zur Berechnung der Gewichtsverteilung eines binären linearen Blockcodes eine sehr lange Schleife als Pipeline ausgeführt. Sie enthält zwei komplexe spezielle kombinatorische Funktionen, die binäre Multiplikation mit einer großen Generator-Matrix und die Gewichtsberechnung der Code-Worte.

9.1.2 Speicherzugriff in Pipelines

In Abschnitt 5.2.1 wurde die Einschränkung getroffen, daß Zuweisungen in einem Pipeline-Kandidaten nur an Vektoren und skalare Variablen erfolgen dürfen. Wir wollen nun betrachten, welche Konsequenzen sich für die Vektorisierung und Pipeline-Synthese ergeben, wenn Zuweisungen an allgemeine Felder (die also keine Vektoren sind) mit beliebigen Indexausdrücken erlaubt sind. Für die Vektoren bleibt dabei aber die Beschränkung auf lineare Indexfunktionen bestehen.

Da die Feldelemente durch im Schleifenrumpf berechnete Zwischenergebnisse adressiert werden, besteht zwischen jedem Schreib- und Lesezugriff eines Feldes eine potentielle Abhängigkeit. Die Abhängigkeits-Analyse führt also zu Rückkopplungszyklen wie bei skalaren Variablen. Das Lesen, Verarbeiten und Schreiben eventuell mehrerer Felder muß in einem Pipeline-Takt erfolgen. Jedoch können Felder nicht einfach in Registern gespeichert werden, und an Felder zugewiesene Zwischenergebnisse (innerhalb einer Schleifeniteration) nicht durch einen Bus mit dem nächsten Operator verbunden werden, da sich ein Schreibzugriff auf ein anderes Feldelement als der nachfolgende Lesezugriff beziehen kann.

Deshalb muß folgendermaßen vorgegangen werden: Für jedes Feld wird Speicher (RAM) allokiert, der sich nicht in der gleichen Speicherbank wie die Vektoren befinden darf, um einen gleichzeitigen Zugriff zu ermöglichen.[1] Für das RAM werden Steuersignale synthetisiert, um die Felder in derselben Reihenfolge wie in der ursprünglichen Schleife zu lesen und zu schreiben. Dies erfordert interne Zustände innerhalb eines Pipeline-Taktes, wodurch die Takte länger werden und die PCU komplizierter und auch programmabhängig wird. Natürlich werden — wie bei Vektoren — auch alle referenzierten allgemeinen Felder vor der Pipeline-Ausführung zur RE und danach zurück zum Wirt kopiert.

Wegen der mit der Zahl der Felder zunehmenden Komplexität der zusätzlich nötigen Schaltungselemente ist für eine effiziente Implementierung die Beschränkung auf

[1] Da der Speicherzugriff die Leistungsfähigkeit der Pipeline wesentlich beeinflußt, eignet sich dafür nur schneller Speicher.

ein allgemeines Feld, das in dem Schleifenrumpf nur einmal gelesen und geschrieben wird, sinnvoll. Dann sind nur wenige zusätzliche interne Pipeline-Zustände und Steuersignale notwendig. Dies ist etwa bei dem bereits erwähnten Beispiel aus Anhang A der Fall. In der Schaltung wurde für das allgemeine Feld eine eigene Speicherbank durch das in der Xilinx-4000-Serie verfügbare sehr schnelle on-chip RAM realisiert. So konnte trotz der zusätzlichen RAM-Ansteuerung derselbe Pipeline-Takt (150 *ns*) wie bei einer einfachen Pipeline mit einem Eingabe- und einem Ausgabe-Port erreicht werden. Wenn auf dem verwendeten FPGA on-chip RAM implementiert werden kann, ist der zusätzliche Speicherzugriff in der Pipeline also auch in unserem Maschinenmodell mit nur einer externen (off-chip) Speicherbank möglich.

9.2 Schleifen-Transformationen

Neben den auf niedrigerer Ebene ansetzenden Synthese-Erweiterungen können auch die von klassischen Vektorisierern bekannten komplexeren Schleifen-Transformationen auf Quellsprach-Ebene (vgl. Abschnitt 4.4.2) zur Erweiterung und Flexibilisierung unseres Verfahrens eingesetzt werden. Für einen Pipeline-Übersetzer sind vor allem die Schleifen-Aufteilung und der Schleifen-Austausch für die im folgenden genannten Optimierungen relevant. Es müssen jedoch Kriterien entwickelt werden, anhand derer ein Übersetzer entscheidet, ob sich eine Transformation lohnt.

9.2.1 Vektorisierung der längsten Schleife

Um effiziente Pipelines zu synthetisieren, sollten möglichst lange Schleifen verwendet werden. Ist aber eine innere Schleife sehr kurz (oder gar nicht vektorisierbar), so kann in einigen Fällen durch Schleifen-Austausch eine wesentlich längere Schleife vektorisiert werden.

9.2.2 Behebung von Hardware-Engpässen

```
FOR I := 1 TO N DO
 FOR J := 1 TO L*M DO
  F(X[I],Y[J]);
 END
END
```

(a)

```
FOR I := 1 TO N DO
 FOR K := 1 TO L DO
  FOR J := 1 TO M DO
   F(X[I],Y[(K-1)*M+J]);
  END
 END
END
```

(b)

```
FOR K := 1 TO L DO
 FOR I := 1 TO N DO
  FOR J := 1 TO M DO
   F(X[I],Y[(K-1)*M+J]);
  END
 END
END
```

(c)

Abbildung 9.1: Transformation zur Behebung eines Hardware-Engpasses

Abbildung 9.1 zeigt eine Transformation, die angewendet werden kann, wenn die verfügbare Hardware für eine Pipeline nicht ausreicht. Da innere Schleifen eines Pipeline-Kandidaten ausgerollt werden, bestimmt ihre Länge die Zahl der später nötigen Hardware-Operatoren. Dies ist etwa bei der zu vektorisierenden I-Schleife in Teilbild (a) der Fall. Deshalb teilt die Transformation von (a) nach (b) zunächst die innere J-Schleife in die K-Schleife und eine kürzere J-Schleife auf. Danach vertauscht die Transformation von (b) nach (c) die I-Schleife und die K-Schleife. So gelangt die neue Schleife ganz nach außen und wird somit unabhängig von der Vektorisierung. Dafür ist die innere J-Schleife jetzt kürzer, während die Länge der jetzt mittleren, zu vektorisierenden I-Schleife unverändert bleibt. Bei der Normalisierung wird also die ursprüngliche innere Schleife nur partiell aufgerollt, was in dem reduzierten Hardware-Bedarf resultiert.

Da die Aufteilung der inneren Schleife in das Produkt L*M nicht vorgegeben ist, sind bei dieser Transformation verschiedene Flächen/Geschwindigkeits-Kompromisse (bzw. Flächen/Parallelitäts-Kompromisse) möglich. Und falls die ursprüngliche Länge der inneren Schleife nicht in geeignete Faktoren zerlegbar ist, kann sie künstlich verlängert werden, indem zusätzliche "neutrale" Werte für die Berechnung F zugefügt werden. (Bei einem FIR-Filter wären dies beispielsweise Koeffizienten mit dem Wert 0, die das berechnete Ergebnis nicht verändern.) Dieses Vorgehen ermöglicht auch die Behandlung von Kandidaten mit variabler innerer Schleifenlänge (d. h. Pipeline-Länge), die von dem ursprünglichen Verfahren nicht behandelt werden können.

9.2.3 Behebung von lokalen Speicher-Engpässen

Wenn der lokale Speicher für alle in der Schleife bearbeiteten Daten zu klein ist, kann ein einfacheres Verfahren verwendet werden: Durch Aufteilen der Kandidaten-Schleife in mehrere kleinere Blöcke wird auch der zur Pipeline-Ausführung benötigte lokale Speicher kleiner.

Allerdings muß dazu genau bestimmt werden, welche Teile eines Feldes von einer Schleife definiert und benutzt werden. Eine solche elementgenaue Analyse der Datenabhängigkeiten könnte auch prinzipiell von allen Koprozessoren genutzt werden, um unnötige Übertragungen einzusparen.

9.2.4 Auflösung irregulärer Abhängigkeiten

Schließlich können wir irreguläre, schleifengetragene Vektorabhängigkeiten (vgl. Abschnitt 5.3.1) durch eine Transformation auflösen: Werden die Iterationen, zwischen denen eine Abhängigkeit besteht, explizit bestimmt, dann kann die Schleife so in einzelne Blöcke aufgeteilt werden, daß innerhalb eines Blocks keine irregulären Abhängigkeiten bestehen. Folglich können die Blöcke einzeln vektorisiert werden, wodurch wieder für mehr Fälle eine Pipeline-Synthese möglich wird.

9.3 Erweiterung der Partitionierung

Auch für das vorgeschlagene Partitionierungs-Verfahren sind mehrere Erweiterungen möglich.

Bei genügend Hardware-Ressourcen können mehrere Pipelines auf einem FPGA zusammengefaßt werden, damit nicht zwischen jeder Pipeline umkonfiguriert werden muß. Dies ist allerdings nur sinnvoll, wenn es nicht durch eine Verringerung der Parallelität der einzelnen Pipelines erkauft werden muß.

Die Ansteuerung kann von der PCU übernommen werden, indem die Eingaben und Steuersignale an alle Pipelines verteilt werden. Von allen Pipeline-Ausgaben werden dann durch Multiplexer (oder durch einen Tri-State-Bus) die richtigen ausgewählt. Die Auswahl wird durch ein vom Wirt gesetztes Register gesteuert. Der Hardware-Bedarf ergibt sich aus der Größe einer PCU, der einzelnen Pipelines und der zusätzlich benötigten Multiplexer.

Bei mehreren Pipelines müssen Verfahren zur Bestimmung der günstigsten Kombination für eine Konfiguration entwickelt werden, die durch Kombination von nacheinander aufgerufenen Pipelines die Zahl der erforderlichen Rekonfigurierungen so weit wie möglich reduzieren. Bei mehreren oder partiell rekonfigurierbaren FPGAs muß die Partitionierung auch festlegen, wann welche FPGA(-Teile) rekonfiguriert werden. Auch eine (eventuell spekulative) Vorkonfigurierung der gerade nicht benötigten FPGAs ist denkbar, um Ausführungszeit einzusparen.

9.4 Verallgemeinerung des Rechner-Modells

Das in dieser Arbeit entwickelte Verfahren ist prinzipiell auf verschiedenartigen SP-Rechner-Architekturen anwendbar, denn die Pipeline-Synthese ist von der verwendeten FPGA-Architektur unabhängig, und die vorausgesetzten Laufzeit-Funktionen zur Datenübertragung in den lokalen Speicher können bei direkter Kopplung mit einem DMA-Kanal oder – wie beim EVCI – indirekt unter Einbeziehung von FPGA-Ressourcen implementiert werden. Jedoch treten bei komplexeren Architekturen zusätzliche Schwierigkeiten und auch neue Optimierungsmöglichkeiten auf.

9.4.1 Mehrere Speicherbänke

Sind auf einem SP-Rechner mehrere Speicherbänke vorhanden, so müssen die von einer Pipeline verarbeiteten Vektoren (und eventuell allgemeinen Felder) auf diese Bänke verteilt werden. Dies sollte so vorgenommen werden, daß auf die Bänke möglichst gleichzeitig zugegriffen werden kann. Beispielsweise könnte der Eingabevektor einer einfachen Pipeline in Speicherbank A und der Ausgabevektor in Bank B liegen. So wird die Pipeline-Zykluszeit T_C verkürzt, da Schreiben und Lesen gleichzeitig erfolgen können.

Jedoch sind manchmal zusätzliche Analysen und Programmtransformationen nötig, um diese Aufteilung vornehmen zu können. Sind zwei Vektoren etwa Teilfelder desselben Feldes, so muß analysiert werden, ob die Teilfelder MAY-Aliase sind, die also überhaupt nicht auf zwei Bänke verteilt werden können. Ist dies nicht der Fall, so muß das Feld (zumindest konzeptionell) in zwei Felder zerlegt werden, damit bei der Adressierung der einzelnen Teilfelder die richtige Speicherbank angesprochen wird.

9.4.2 Multi-FPGA-Architekturen

Besteht eine Architektur aus mehreren FPGAs, muß die Schaltung auf diese verteilt werden.[2] Wegen der begrenzten Zahl der Pins, die zwei FPGAs verbinden, ist dies schwieriger als die Verdrahtung einer Schaltung in *einem* größeren FPGA. Es bestehen allgemeine Ansätze zur Schaltungs-Partitionierung. Sie zu verwenden ist aber nicht optimal, da die spezielle Struktur der Pipelines ausgenützt werden sollte. Das betrifft auch die PCU, die so verändert werden könnte, daß die Partitionierung erleichtert wird. Man könnte etwa eine verteilte PCU entwerfen oder bestimmte Komponenten (z. B. Zähler) redundant auf mehreren FPGAs implementieren, um dadurch Pins zu sparen.

Allerdings kann die Schaltungs-Partitionierung nicht unabhängig von der Speicher-Aufteilung betrachtet werden, da die Speicherbänke typischerweise mit verschiedenen FPGAs verdrahtet sind und so die Partitionierung der Schaltung davon abhängt, welche Daten in welcher Speicherbank liegen. Denn bei größeren Architekturen ist es wegen der Pin-Begrenzung nicht möglich, die Datenleitungen aller Speicherbänke an alle FPGAs zu verteilen.

9.5 Erweiterungen der Eingabesprache

Für viele Anwendungen — insbesondere in der Signal- und Bildverarbeitung — wäre es sehr nützlich, wenn die Eingabesprache über Festkommatypen und -operatoren verfügen würde. Denn in vielen Fällen reichen Festkommawerte aus. Da die meisten Programmiersprachen aber nur Gleitkommatypen zur Verfügung stellen, ist der Anwender gezwungen, diese zu verwenden. Dies behindert aber wiederum die Hardware-Implementierung.

Eine Alternative zur Änderung der Sprache wäre eine automatische Ersetzung von Gleitkommazahlen durch Festkommazahlen, wo dies semantisch äquivalent ist oder das sichtbare Ergebnis (z. B. bei Bildern) nicht beeinträchtigt. Eine solche Ersetzung ist aber kaum automatisch durchzuführen.

Einige Hardware/Software-Codesign-Systeme verwenden Zeitbedingungen in der Eingabesprache. Für einen Pipeline-Compiler wäre dies aber nur dann sinnvoll, wenn er für Echtzeit-Anwendungen in eingebetteten Systemen verwendet werden soll.

[2] engl. chip partitioning

Kapitel 10

Zusammenfassung und Ausblick

Dieses Kapitel faßt die erreichten Ergebnisse zusammen und weist auf zukünftige Weiterentwicklungen hin.

10.1 Zusammenfassung

In dieser Arbeit wurden neue Übersetzungsmethoden für strukturprogrammierbare Rechner vorgestellt. Mit den entwickelten Verfahren können aus einer Spezifikation in einer sequentiellen, imperativen Programmiersprache automatisch die Software- und Hardware-Anteile einer Anwendung sowie die Schnittstelle zwischen beiden generiert werden. Es handelt sich also um einen software-basierten Codesign-Ansatz für eine fest vorgegebene Zielarchitektur, einen Mikroprozessor mit einer FPGA-basierten rekonfigurierbaren Einheit. Dabei wird nicht die Minimierung des Hardware-Bedarfs unter Berücksichtigung konkreter Echtzeitanforderungen, sondern die bestmögliche Ausnutzung einer vorhandenen Hardware-Konfiguration angestrebt.

Dieser software-basierte Ansatz ermöglicht es auch Entwicklern ohne Erfahrung im Hardware-Entwurf, SP-Rechner einzusetzen.

Bisher bekannte derartige Verfahren verwenden die Standard-High-Level-Synthese zur Erzeugung rekonfigurierbarer Koprozessoren aus einer sequentiellen Eingabesprache. Hierbei ist die erreichbare Parallelität jedoch meist auf Operationen innerhalb eines Basisblocks beschränkt. Auch mit den bekannten Erweiterungen (Schleifen-Pipelining) können Schleifen mit Vektor-Zuweisungen nicht optimiert werden.

Ein wesentlicher Fortschritt dieser Arbeit besteht in der Entwicklung einer neuen Methode, die erstmalig aus einer imperativen, sequentiellen Eingabesprache durch eine verallgemeinerte Vektorisierung effiziente, parallele Pipeline-Schaltungen synthetisiert. Deshalb werden alle vektorisierbaren Schleifen als Hardware-Kandidaten betrachtet. Die zur Vektorisierung notwendige Abhängigkeits-Analyse wurde genau an die Erfordernisse der flexiblen FPGA-Koprozessoren angepaßt. Dazu wurde der

Begriff der regulären Abhängigkeiten definiert, die eine Vektorisierung nicht verhindern, da sie in einer Pipeline durch Rückkopplungsregister realisiert werden können. Außerdem wurde für die Pipelines ein auf ganzzahliger linearer Programmierung beruhendes Verfahren zur globalen Register-Minimierung entwickelt, um die knappen FPGA-Flipflops optimal zu nützen.

Der zweite Hauptbeitrag der Arbeit besteht aus der Entwicklung und Integration von automatischen Verfahren zur Ansteuerung der Pipelines mit einer Pipeline-Steuerungseinheit, zur Einbindung der Hardware-Koprozessoren in das auf dem Wirt laufende Programm mit vorgegebenen Laufzeit-Funktionen sowie zur dynamischen Hardware/Software-Partitionierung. Die Verfahren beruhen auf einem allgemeinen SP-Rechner-Modell und einer generischen Schnittstelle, die von maschinenspezifischen Details abstrahieren. Sie sind also von dem konkreten SP-Rechner und der verwendeten FPGA-Familie unabhängig. Die Pipeline-Synthese und das Schnittstellen-Konzept wurden prototypisch im MODULA PIPELINE COMPILER realisiert. Dadurch wurde die Praxistauglichkeit der Verfahren gezeigt.

Die Hardware/Software-Partitionierung verwendet Parameter des Rechner-Modells und wählt die lohnenden Koprozessoren aus. Dazu benötigt sie präzise Vorhersagen des Ressourcen-Bedarfs und der Geschwindigkeit der synthetisierten Hardware. Diese sind für Pipeline-Schaltungen einfach zu gewinnen, da der Hardware-Bedarf nur von der festen Größe der vorgegebenen PCU und von den in der Pipeline verwendeten Operatoren abhängt. Und die Geschwindigkeit wird allein durch die Taktfrequenz bestimmt. Neben der erwarteten Beschleunigung durch die Pipeline selbst müssen zur Partitionierung auch die Kosten für die Konfigurierung der FPGAs und für die Datenübertragung zwischen Wirt und RE berücksichtigt werden. Dazu wurde ein dynamisches Verfahren entwickelt, das zur Laufzeit anhand der aktuellen Daten und des Konfigurationszustands der FPGAs die Ausführung in Software oder Hardware auswählt. Dies ermöglicht auch eine automatische dynamische Umkonfigurierung der FPGAs zwischen größeren Phasen des Programms.

10.2 Ausblick

Die Entwicklung integrierter Übersetzer für strukturprogrammierbare Rechner steckt noch in den Kinderschuhen. Denn es gibt keine einheitliche Meinung darüber, welche Übersetzungs- und Synthesetechniken am besten verwendet werden, und wie sie in einem einheitlichen System integriert werden sollen. Außerdem hat sich noch keine allgemein akzeptierte Architektur für die Rechner selbst herauskristallisiert. Deswegen ist die vorliegende Arbeit nur ein erster Schritt auf einem Weg, dessen Richtung noch gar nicht genau abzusehen ist.

Mögliche Erweiterungen der hier vorgestellten Methodik wurden bereits in Kapitel 9 vorgestellt, das deshalb als ausführlicher Ausblick auf zukünftige Arbeiten betrachtet werden kann. Bei der Entwicklung eines vollständigen Übersetzers sollten die Empfehlungen aus Abschnitt 7.4 berücksichtigt und vor allem die in Abschnitt

9.1 erwähnte Integration mit der Optimierung spezieller kombinatorischer Funktionen implementiert werden. Denn diese sind für eine bessere Ausnutzung der Leistungsfähigkeit der RE entscheidend.

Zur Portierung des Übersetzers auf größere SP-Rechner (vgl. Abschnitt 9.4) sind vor allem weiterführende Arbeiten zur Partitionierung der Schaltung auf mehrere FPGAs sowie zur optimalen Nutzung mehrerer Speicherbänke, die über die FPGAs verteilt sind, nötig. Davon ist auch die Partitionierung betroffen (Abschnitt 9.3), und die Pipeline-Steuerungseinheit muß auf verschiedene FPGAs verteilt oder teilweise dupliziert werden. Ein weiterer Schritt wäre die Verwendung der für arithmetische Berechnungen weitaus geeigneteren wort-orientierten FPGAs.

Schließlich würde die Erweiterung des Übersetzers für andere Eingabesprachen (etwa ANSI-C) oder die vollständige Integration in ein flexibles Compiler-Framework einen breiteren Einsatz der entwickelten Methoden ermöglichen. Dasselbe gilt für die Einbeziehung eingebetteter SP-Rechner, also eine Weiterentwicklung in Richtung Echtzeitsysteme.

Literaturverzeichnis

[AK87] R. Allen und K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, S. 491–542, Oktober 1987.

[AP95] P. Athanas und K. L. Pocek, Hrsg. *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, Napa, CA, April 1995.

[AS93] P. M. Athanas und H. F. Silverman. Processor reconfiguration through instruction-set metamorphosis. *Computer*, 26(3):11–18, März 1993.

[ASU86] A.V. Aho, R. Sethi, und J.D. Ullman. *Compilers — Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Ath92] P. M. Athanas. *An Adaptive Machine Architecture and Compiler for Dynamic Processor Reconfiguration*. Dissertation, Brown University, Mai 1992.

[AW93] U. Aßmann und M. Weinhardt. Interprocedural heap analysis for parallelizing imperative programs. In W. K. Giloi, S. Jähnichen, und B. D. Shriver, Hrsg., *Massively Parallel Programming Models (MPPM)*, S. 74–82. IEEE Computer Society Press, September 1993.

[AWG94] L. Agarwal, M. Wazlowski, und S. Ghosh. An asynchronous approach to efficient execution of programs on adaptive architectures utilizing FPGAs. In *Proc. IEEE Workshop on FPGAs for Custom Computing Machines*, Napa, CA, April 1994.

[BAK96] D. A. Buell, J. M. Arnold, und W. J. Kleinfelder. *Splash 2 - FPGAs in a Custom Computing Machine*. IEEE Computer Society Press, 1996.

[Ber92] M. Berkelaar. Unix manual page of `lp_solve`. Eindhoven University of Technology, Design Automation Section, 1992.

[BP93] D. A. Buell und K. L. Pocek, Hrsg. *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, Napa, CA, April 1993.

[BP94] D. A. Buell und K. L. Pocek, Hrsg. *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, Napa, CA, April 1994.

[BRV89] P. Bertin, D. Roncin, und J. Vuillemin. Introduction to programmable active memories. In J. McCanny, J. McWhirther, und E. Swartslander Jr., Hrsg., *Systolic Array Processors*, S. 300–309. Prentice Hall, 1989.

[BRV93a] P. Bertin, D. Roncin, und J. Vuillemin. Programmable active memories: a performance assessment. In G. Borriello und C. Ebeling, Hrsg., *Research on Integrated Systems: Proceedings of the 1993 Symposium*, S. 88–102, 1993.

[BRV93b] P. Bertin, D. Roncin, und J. Vuillemin. Programmable active memories: a performance assessment. Technischer Bericht 3, DEC Paris Research Lab, März 1993.

[BT94] P. Bertin und H. Touati. PAM programming environments: Practice and experience. In D. A. Buell und K. L. Pocek, Hrsg., *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, S. 133–138, Napa, CA, April 1994.

[Cam90] R. Camposano. From behavior to structure: High-level synthesis. *IEEE Design & Test of Computers*, S. 8–19, Oktober 1990.

[Cas93] S. Casselman. Virtual computing and the virtual computer. In D. A. Buell und K. L. Pocek, Hrsg., *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, S. 43–48, Napa, CA, April 1993.

[CTS95] S. Casselman, M. Thornburg, und J. Schewel. Creation of hardware objects in a reconfigurable computer. In *Field Programmable Logic and Applications; 5th International Workshop*, Oxford, UK, September 1995. Springer-Verlag.

[CW91] R. Camposano und W. Wolf. *High-Level VLSI Synthesis*. Kluwer Academic Publishers, 1991.

[DeH94] A. DeHon. DPGA-coupled microprocessors: Commodity ICs for the early 21st century. In D. A. Buell und K. L. Pocek, Hrsg., *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, S. 31–39, Napa, CA, April 1994.

[ECP96] C. Ebeling, D. C. Cronquist, und P. Franklin. RaPiD - reconfigurable pipelined datapath. In *Field-Programmable Logic and Applications; 6th International Workshop*, Darmstadt, Germany, September 1996. Springer-Verlag.

[EHB93] R. Ernst, J. Henkel, und Th. Benner. Hardware-Software Cosynthesis for Microcontrollers. *IEEE Design & Test of Computers*, S. 64–75, Dezember 1993.

[Gal95] D. Galloway. The transmogrifier C hardware description language and compiler for FPGAs. In P. Athanas und K. L. Pocek, Hrsg., *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, S. 136–144, Napa, CA, April 1995.

[Gao89] G. R. Gao. Algorithmic aspects of balancing techniques for pipelined data flow code generation. *Journal of Parallel and Distributed Computing*, 6:39–61, 1989.

[GDWL92] D. D. Gajski, N. D. Dutt, A. Wu, und St. Lin. *High-Level Synthesis: Introduction to Chip and System Design.* Kluwer Academic Publishers, 1992.

[GG93] S. A. Guccione und M. J. Gonzalez. A data-parallel programming model for reconfigurable architectures. In *Proc. IEEE Workshop on FPGAs for Custom Computing Machines*, Napa, CA, April 1993.

[GG95a] S. Guccione und M. J. Gonzalez. Classification and performance of reconfigurable architectures. In *Field Programmable Logic and Applications; 5th International Workshop*, Oxford, UK, September 1995. Springer-Verlag.

[GG95b] S. Guccione und M. J. Gonzalez. Supercomputing with reconfigurable architectures. In *Field Programmable Logic and Applications; 5th International Workshop*, Oxford, UK, September 1995. Springer-Verlag.

[GH92] H. Grunbacher und R. Hartenstein, Hrsg. *FPGAs: Proceedings of the 1992 International workshop on field-programmable logic and applications*, Vienna, Austria, September 1992. Spinger-Verlag.

[GHK+91] M. Gokhale, W. Holmes, A. Kopser, S. Lucas, R. Minnich, D. Sweely, und D. Lopresti. Building and using a highly parallel programmable logic array. *IEEE Computer*, 24(1):81–89, Januar 1991.

[GKLM90] M. Gokhale, A. Kopser, S. Lucas, und R. Minnich. The logic description generator. Technischer Bericht SRC-TR-90-011, Supercomputing Research Center, 1990.

[GM95] M. Gokhale und A. Marks. Automatic synthesis of parallel programs targeted to dynamically reconfigurable logic arrays. In *Field Programmable Logic and Applications; 5th International Workshop*, Oxford, UK, September 1995. Springer-Verlag.

[Guc] S. Guccione. List of FPGA-based Computing Machines. http://www.io.com/~guccione/HW_list.html.

[Guc95] S. Guccione. *Programming Fine-Grained Reconfigurable Architectures.* Dissertation, University of Texas at Austin, Mai 1995.

[Har95] R. Hartenstein. Das aktuelle Schlagwort: Custom Computing Machines. *Informatik-Spektrum*, 3:228–229, August 1995.

[Har96] R. W. Hartenstein. High Performance Computing: über Szenen und Krisen. In *Custom Computing, GI/ITG Workshop*, Schloß Dagstuhl, Germany, Juni 1996. Chemnitzer Informatik-Berichte.

[HBH+96] R. W. Hartenstein, J. Becker, M. Herz, R. Kress, und U. Nageldinger. A partitioning programming environment for a novel parallel architecture. In *10th International Parallel Processing Symposium*, Honolulu, Hawaii, März 1996.

[HBK96] R. W. Hartenstein, J. Becker, und R. Kress. Custom computing machines vs. hardware/software co-design: From a globalized point of view. In *Field-Programmable Logic and Applications; 6th International Workshop*, Darmstadt, Germany, September 1996. Springer-Verlag.

[HE95] U. Holtmann und R. Ernst. Combining MBP-speculative computation and loop pipelining in high-level synthesis. In *IEEE/ACM Proc. of EDAC'95*, S. 550–556, 1995.

[HG96] R. W. Hartenstein und M. Glesner, Hrsg. *Proc. of 6th International Workshop on Field Programmable Logic and Applications*, Darmstadt, Germany, September 1996. Springer-Verlag.

[HKR94] R. Hartenstein, R. Kress, und H. Reinig. A new FPGA architecture for word-oriented datapaths. In *Field-Programmable Logic; 4th International Workshop*, Prague, Czech Republic, September 1994. Springer-Verlag.

[Hol95] U. Holtmann. *Synthese von Co-Prozessoren mit spekulativer Ausführung*. Dissertation, Technische Universität Braunschweig, 1995.

[HS94] R. Hartenstein und M. Servit, Hrsg. *Proc. of 4th International Workshop on Field Programmable Logic and Applications*, Prague, Czech Republic, September 1994. Springer-Verlag.

[HW95] B. L. Hutchings und M. J. Wirthlin. Implementation approaches for reconfigurable logic applications. In *Field-Programmable Logic and Applications; 5th International Workshop*, Oxford, UK, September 1995. Springer-Verlag.

[HX88] K. Hwang und Z. Xu. Multipipeline networking for compound vector processing. *IEEE Transactions on Computers*, 37:33–47, Januar 1988.

[IEE] IEEE. *IEEE Standard VHDL Language Reference Manual*. IEEE STD-1076.

[IS93] C. Iseli und E. Sanchez. Spyder: A reconfigurable VLIW processor using FPGAs. In D. A. Buell und K. L. Pocek, Hrsg., *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, S. 17–24, Napa, CA, April 1993.

[JEO+94a] A. Jantsch, P. Ellervee, J. Öberg, A. Hemani, und H. Tenhunen. Hardware/software partitioning and minimizing memory interface traffic. In *Proc. of European Design Automation Conf. '94*. IEEE Computer Society Press, September 1994.

[JEO+94b] A. Jantsch, P. Ellervee, J. Öberg, A. Hemani, und H. Tenhunen. A software oriented approach to hardware/software codesign. In *Proc. of the Poster Session of CC'94 - Internat. Conf. on Compiler Construction*, Technical Report, Dept. of Comp. and Inform. Science, Linköping Univ., Sweden, April 1994.

[KKT+95] T. Kuhn, U. Kebschull, P. Thole, E. Schubert, und W. Rosenstiel. Bibliotheksorientierte Technologieabbildung synthetisierter Datenpfade. In *2. GI/ITG Workshop Anwenderprogrammierbare Schaltungen*, S. 105–111. FZI-Publikation 2/95, Juni 1995.

[KLB95] H. Kalouti, D. E. Lazic, und T. Beth. On the relation between distance distributions of binary block codes and the binomial distribution. *Annales des Télécommunications*, 50(9–10):762–778, 1995.

[KM96] H. Kalouti und A. Mathias. On computing weight distributions of binary linear blockcodes. Preprint, 1996.

[KW95] T. Kean und B. Wilkie. The XC6200 FastMap processor interface. In *Field Programmable Logic and Applications; 5th International Workshop*, Oxford, UK, September 1995. Springer-Verlag.

[Len90] Th. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. Teubner/John Wiley & Sons, 1990.

[Lin92] J. Lindemeyer. Einteilung und Verteilung von Parallelität bei Schleifen über Listen. Diplomarbeit, Universität Karlsruhe, Februar 1992.

[LRS83] C. E. Leiserson, F. M. Rose, und J. B. Saxe. Optimizing synchronous circuitry by retiming. In *Proc. Third Caltech Conference on VLSI*. Springer-Verlag, März 1983.

[LS83] C. E. Leiserson und J. B. Saxe. Optimizing synchronous systems. *Journal of VLSI and Computer Systems*, 1:41–67, 1983.

[LSU89] R. Lipsett, C. F. Schaefer, und C. Ussery. *VHDL: Hardware Description and Design*. Kluwer Academic Publishers, 1989.

[ML91] W. Moore und W. Luk, Hrsg. *FPGAs: Proceedings of the 1991 International workshop on field-programmable logic and applications*, Oxford, England, September 1991. Abingdon EE and CS Books.

[ML93] W. Moore und W. Luk, Hrsg. *More FPGAs: Proceedings of the 1993 International workshop on field-programmable logic and applications*, Oxford, England, September 1993. Abingdon EE and CS Books.

[ML95] W. R. Moore und W. Luk, Hrsg. *Proc. of 5th International Workshop on Field Programmable Logic and Applications*, Oxford, UK, September 1995. Springer-Verlag.

[MNT+94] T. Miyazaki, H. Nakada, A. Tsutsui, K. Yamada, und N. Ohta. A speed-up technique for synchronous circuits realized as LUT-based FPGAs. In *Field Programmable Logic and Applications; 4th International Workshop*, Prague, Czech Republic, September 1994. Springer-Verlag.

[MR92] P. Marwedel und W. Rosenstiel. Synthese von Register-Transfer-Strukturen aus Verhaltensbeschreibungen. *Informatik-Spektrum*, S. 5–22, Februar 1992.

[PA96] K. L. Pocek und J. Arnold, Hrsg. *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, Napa, CA, April 1996.

[PL91] I. Page und W. Luk. Compiling occam into FPGAs. In *FPGAs. International Workshop on Field Programmable Logic and Applications*, S. 271–283, Oxford, UK, September 1991.

[Sch88a] Friedrich Wilhelm Schröer. Das GMD Modula-2 Entwicklungssystem. *GMD-Spiegel*, 1, 1988.

[Sch88b] Friedrich Wilhelm Schröer. Mobil: An intermediate language for portable optimizing compilers. Internes Papier, GMD, 1988.

[Syna] Synopsys, Mountain View, CA. *Design Compiler Family Reference Manual.*

[Synb] Synopsys, Mountain View, CA. *VHDL Compiler Reference Manual.*

[TCS94] M. Thornburg, S. Casselman, und J. Schewel. *Engineers' Virtual Computer Users Guide — EVCUs.* Virtual Computer Corporation, 1994.

[TIW94] H. W. Thimbleby, S. Inglis, und I. H. Witten. Displaying 3D images: Algorithms for single image random-dot stereograms. *Computer*, S. 38–48, Oktober 1994.

[VBR+96] J. E. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. H. Touati, und Ph. Boucard. Programmable active memories: Reconfigurable systems come of age. *IEEE Transactions on VLSI Systems*, 4(1):56–69, 1996.

[VG95] F. Vahid und D. Gajski. Incremental hardware estimation during hardware/software partitioning. *IEEE Trans. on VLSI Systems*, S. 459–464, September 1995.

[Wak94] J. F. Wakerly. *Digital Design Principles and Practices.* Prentice–Hall, Inc., 1994.

[WBO95] A. Wenban, G. Brown, und J. O'Leary. Developing interface libraries for reconfigurable data acquisition boards. In *Field Programmable Logic and Applications; 5th International Workshop*, Oxford, UK, September 1995. Springer-Verlag.

[Wei92] M. Weinhardt. Haldenanalyse für Modula-2. Diplomarbeit, Universität Karlsruhe, Juli 1992.

[Wei95] M. Weinhardt. Integer programming for partitioning in software oriented codesign. In *Field-Programmable Logic and Applications; 5th International Workshop*, Oxford, UK, September 1995. Springer-Verlag.

[Wei96a] M. Weinhardt. CCM-Programmierung mit Pipeline-Parallelität. In *Custom Computing, GI/ITG Workshop*, Schloß Dagstuhl, Germany, Juni 1996. Chemnitzer Informatik-Berichte.

[Wei96b] M. Weinhardt. Computing weight distributions of binary linear block codes on a CCM. In *Field-Programmable Logic and Applications; 6th International Workshop*, Darmstadt, Germany, September 1996. Springer-Verlag.

[Wei96c] M. Weinhardt. Portable pipeline synthesis for FCCMs. In *Field-Programmable Logic and Applications; 6th International Workshop*, Darmstadt, Germany, September 1996. Springer-Verlag.

[Wei97a] M. Weinhardt. Compilation and pipeline synthesis for reconfigurable architectures. In *Reconfigurable Architectures - High Performance by Configware; Proc. RAW'97.* ITpress Verlag, April 1997.

[Wei97b] M. Weinhardt. Pipeline synthesis and optimization for reconfigurable custom computing machines. Interner Bericht 1/97, Universität Karlsruhe, Fakultät für Informatik, 1997.

[WOB93] A. S. Wenban, J. W. O'Leary, und G. M. Brown. Codesign of communication protocols. *Computer*, S. 46–52, Dezember 1993.

[Wol96] M. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, 1996.

[Xil94a] Xilinx, San Jose, CA. *The Programmable Logic Data Book*, 1994.

[Xil94b] Xilinx, San Jose, CA. *XACT User Guide*, 1994.

[Xil94c] Xilinx, San Jose, CA. *XACT Xilinx Synopsys Interface FPGA User Guide*, 1994.

[ZC91] H. Zima und B. Chapman. *Supercompilers for Parallel and Vector Computers*. Addison-Wesley, 1991.

Verzeichnis der Abkürzungen

ASIC
Application-Specific Integrated Circuit (S. 14)

CCM
Custom Computing Machine (S. 18)

EVC1
Engineer's Virtual Computer 1 (S. 18)

FIR-Filter
Finite Impulse Response Filter (S. 106)

FPGA
Field-Programmable Gate Array (S. 12)

ILP
Integer Linear Program / ganzzahliges lineares Programm (S. 30)

MOBIL
MOdula Backend Interface Language (S. 95)

MOCKA
MOdula Compiler KArlsruhe (S. 94)

PCU
Pipeline Control Unit / Pipeline-Steuerungseinheit (S. 79)

PLA
Programmable Logic Array (S. 11)

RE
Rekonfigurierbare Einheit (S. 18)

SP-Rechner
Strukturprogrammierbarer Rechner (S. 18)

VHDL
VHSIC Hardware Description Language (S. 14)

Anhang A

Berechnung der Gewichtsverteilung linearer Blockcodes auf dem EVC1

Dieser Anhang beschreibt die Implementierung eines Algorithmus zur Berechnung der Gewichtsverteilung eines binären linearen Blockcodes auf dem EVC1. Wegen der exponentiellen Zeitkomplexität dieses Problems werden schnelle Implementierungen benötigt, wofür die Verwendung rekonfigurierbarer Hardware sehr interessant ist: Unsere Implementierung ist etwa siebenmal schneller als die beste bekannte Software-Lösung und berechnet alle 150 Nanosekunden ein neues Gewicht. Damit können für größere Codes mehrere Tage Rechenzeit eingespart werden. Diese hohe Geschwindigkeit wird erreicht, indem die ganze Berechnung in einer tiefen Pipeline durchgeführt und somit die feinkörnige Parallelität der Hardware ausgenutzt wird. Außerdem beruht die Implementierung auf codespezifischen Logik-Optimierungen, die bei Verwendung des Standard-Befehlssatzes eines Mikroprozessors nicht möglich sind. Die Schaltung wurde in VHDL spezifiziert.

Im nächsten Abschnitt werden binäre lineare Blockcodes und ihre Gewichtsverteilung kurz eingeführt. Danach stellt Abschnitt A.2 die wichtigsten Eigenschaften unserer Implementierung vor, und Abschnitt A.3 präsentiert einige Laufzeitmessungen, die die erreichte Leistungssteigerung zeigen. Schließlich diskutieren wir in Abschnitt A.4, wie die Schaltung automatisch aus einer höheren Programmiersprache (statt aus VHDL) generiert werden könnte.

Die hier vorgestellte Implementierung wurde bereits in [Wei96b] veröffentlicht.

A.1 Gewichtsverteilung linearer Blockcodes

Binäre Blockcodes sind $[N, K]$-Codes, die einen K-Bit-Block (das *Informationswort*) auf ein N-Bit-*Codewort* abbilden. Da $N > K$ gilt, enthält ein Codewort $N - K$ redundante Bits. Der Grad der Redundanz wird durch die *Coderate* $R = \frac{K}{N}$ bestimmt.

Da nicht alle N-Bit-Wörter gültige Codewörter sind, ist es möglich, Fehler zu erkennen und zu korrigieren. Die Qualität eines binären Blockcodes wird durch die Wahrscheinlichkeit eines nicht entdeckten Fehlers bestimmt. Und diese hängt von der minimalen Hamming-Distanz zweier beliebiger Codewörter ab.

Wir betrachten hier nur *lineare Codes*, also Codes, die durch eine binäre $K \times N$ Generatormatrix dargestellt werden können. (Ein Codewort c wird dann durch die Matrizenmultiplikation $c = i \cdot G$ modulo 2 berechnet.) Glücklicherweise ist für lineare Codes die minimale Distanz zweier Codewörter gleich der minimalen Distanz eines beliebigen Wortes zu $\vec{0}$. Diese Distanz, die der Anzahl der Einsen in dem Wort entspricht, wird *Gewicht* des Wortes genannt. Folglich müssen zur Beurteilung der Qualität eines Codes statt der Distanzen nur die Gewichte der Codewörter betrachtet werden.

Die *Gewichtsverteilung* $\{W(r) \mid 0 \leq r \leq N\}$ eines $[N, K]$-Codes beschreibt die Zahl der Codewörter mit dem Gewicht r. Aus ihr kann natürlich das minimale Gewicht abgelesen werden. Aber die Verteilung ist auch von theoretischem Interesse, da sie Aussagen über die asymptotische Optimalität einer Codefamilie ermöglicht. Um $W(r)$ jedoch zu bestimmen, müssen die Gewichte aller 2^K Codewörter berechnet werden. Wegen dieser exponentiellen Komplexität ist die Berechnung der Gewichtsverteilung bereits für relativ kleine Codes ein schwieriges Problem. Effiziente Algorithmen existieren nur für Codes mit besonderen Eigenschaften. Und für Codes mit $K > N - K$ können die *MacWilliams-Identitäten* benutzt werden, um den Rechenaufwand zu reduzieren. Dabei wird die Gewichtsverteilung des im allgemeinen kleineren dualen $[N, N-K]$-Codes $C^{\perp}$ berechnet, um die Verteilung des $[N, K]$-Codes C zu erhalten. Folglich liegt die schwierigste Situation dann vor, wenn $K = N - K$, d. h. $R = 0.5$, gilt. Deshalb betrachten wir nur diesen Fall und nehmen keine zusätzlichen besonderen Eigenschaften der Codes an.

A.2 Implementierung

Wir implementierten eine Gewichtsverteilungs-Schaltung mit tiefer Pipeline auf einer Sun SPARCstation 10 mit der EVC1-Karte der Virtual Computer Corporation (vgl. Abschnitt 7.1). Sie berechnet zunächst für alle Informationsworte i das Codewort $c = i \cdot G$ modulo 2. Dann wird das Gewicht von c bestimmt und ein Zähler für dieses Gewicht erhöht.

Leider muß die FPGA-Schaltung für jeden Code, den wir bewerten wollen, teilweise neu synthetisiert sowie nochmals plaziert und verdrahtet werden. Dies ist wegen der sehr begrenzten rekonfigurierbaren Ressourcen auf der EVC1-Karte nötig. Sie reichen nicht aus, um die Generatormatrix des Codes in den FPGA-Flipflops auf der Karte zu speichern. Vielmehr muß sie in der VHDL-Beschreibung eine Konstante sein, damit eine Logik-Optimierung möglich ist. Obwohl die Änderung der Schaltung für jedes Problem mehr als eine Stunde dauern kann, lohnt sich dieser Aufwand, da für größere Codes *Tage* an Rechenzeit eingespart werden können (s. Abschnitt A.3).

Die Schaltung wurde mit dem Synopsys FPGA Compiler und Xilinx XACT entwickelt. Der ganze Entwurf wurde in VHDL modelliert und mit dem Synopsys VHDL Compiler verarbeitet. Jedoch mußten detaillierte Zeitbedingungen für die XNF-Signale manuell spezifiziert werden, damit das PPR-Programm von Xilinx (Plazierung und Verdrahtung) eine schnelle Schaltung erzeugt. Da aber der Takt des EVC1 nicht an die von PPR erreichten Signalverzögerungen angepaßt werden kann, mußte die Ablaufplanung in dem VHDL-Modell geändert werden, um die von PPR bestimmten Verzögerungen an die vorhandenen Taktfrequenzen anzupassen. Dadurch wurden zeitaufwendige Wiederholungen des Entwurfszyklus erforderlich.

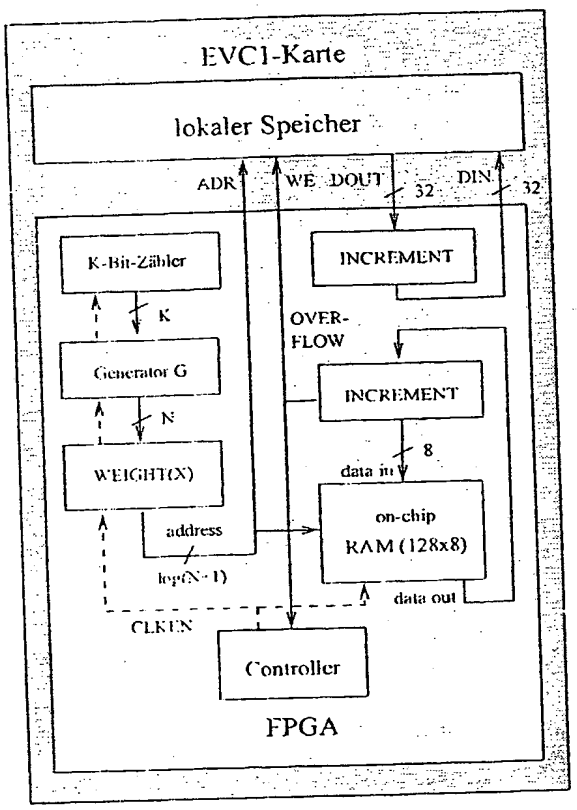


Abbildung A.1: Gewichtsverteilungs-Schaltung

Abbildung A.1 zeigt den Aufbau der Gewichtsverteilungs-Schaltung. Sie wurde für $[N, K]$-Codes mit $N < 128$ und $K \leq 64$ entworfen. Der K-Bit-Zähler erzeugt zunächst alle möglichen Bitmuster (d. h. Informationswörter) und speist den Generator damit. Dieser Block implementiert die Modulo-2-Multiplikation mit der Ge-

neratormatrix G. Da der Generator nur XOR-Gatter enthält, kann er gut logisch optimiert werden und effizient (d. h. platzsparend) auf FPGAs implementiert werden. Danach berechnet der WEIGHT-Block das Gewicht aller N-Bit-Codewörter. Schließlich werden Zähler für die entsprechenden Gewichte erhöht.

Das grundlegende Entwurfsziel für diese Schaltung war, den Durchsatz zu maximieren. Dies kann durch Pipelining für den Generator- und den WEIGHT-Block praktisch unbegrenzt erreicht werden. Auch der einfache Zähler zur Erzeugung der Informationswörter kann effizient implementiert werden. Der Gesamtdurchsatz wird aber durch die Speicher/Inkrementierer-Rückkopplungsschleife zur Erhöhung der Gewichts-Zähler begrenzt. Deshalb wird für die Gewichts-Zähler der schnelle on-chip RAM des Xilinx XC4013 FPGA verwendet (rechts unten in Abbildung A.1). Wegen der begrenzten FPGA-Ressourcen und der von der Wortlänge abhängenden Verzögerung eines Inkrementierers werden jedoch nur die acht niedrigstwertigen Bits der Zähler in diesem RAM gespeichert. Ein Überlauf (OVERFLOW-Signal in Abbildung A.1) eines Zählers bewirkt, daß ein anderer Zähler in dem externen lokalen Speicher auf der EVC1-Karte erhöht wird (Rückkopplungszyklus mit zweitem INCREMENT-Block oben rechts in Abbildung A.1). Während des Zugriffs auf den langsamen externen Speicher wird die Pipeline angehalten. Da diese Ausnahmebehandlung jedoch nur bei einem Überlauf durchgeführt wird, verringert sich der mittlere (amortisierte) Durchsatz der Schaltung kaum.

A.3 Laufzeitmessungen

Wegen des schnellen on-chip RAM und der schnellen Inkrementierer, welche die "dedicated carry logic" des XC4013 einsetzen, erwarteten wir für den Lese-Inkrementiere-Schreib-Zyklus eine Signalverzögerung von höchstens 75 ns. Obwohl die Plazierung und Verdrahtung durch genaue Zeitbedingungen gesteuert wurde, erhöhten hohe Verdrahtungs-Verzögerungen die Zykluszeit jedoch auf 150 ns. Der Durchsatz der Pipeline ist also etwa 6.7 Millionen Gewichtsberechnungen pro Sekunde. Folglich kann die Gewichtsverteilung eines $[70, 35]$-Codes (2^{35} Gewichte) in weniger als 1.5 Stunden berechnet werden. Zur Berechnung derselben Verteilung in Software wären mindestens 9.5 Stunden nötig, da die beste uns bekannte sequentielle Software-Implementierung (vgl. [KM96, KLB95]) eine Mikrosekunde pro Gewicht benötigt. Für diese Implementierung wurde der Algorithmus auf einer Sun SPARCstation 10 mit großem Cache-Speicher manuell optimiert. Der Cache enthält dabei eine vorberechnete Tabelle der Gewichte aller 16-Bit-Worte.

In Tabelle A.1 werden die Laufzeiten unserer SP-Rechner-Implementierung mit der erwähnten Software-Implementierung verglichen. Die relative und absolute Beschleunigung für $K = 30$, 35 und 40 wird angegeben.[1] Die Werte schließen jedoch die Zeit zur Synthese des FPGA-Bitstroms nicht ein.

[1] Die EVC1-Laufzeit wurde für $K = 30$ und $K = 35$ tatsächlich gemessen, aber noch nicht für $K = 40$.

K	Iterationen (2^K)	Laufzeit EVC1	Laufzeit Software	relative Beschleunigung	absolute Beschleunigung
30	$\approx 1 \times 10^9$	2.7 Min.	17.9 Min.	6.7	15.2 Min.
35	$\approx 34 \times 10^9$	1.4 Std.	9.5 Std.	6.7	8.1 Std.
40	$\approx 1 \times 10^{12}$	1.9 Tage	12.7 Tage	6.7	10.8 Tage

Tabelle A.1: Laufzeit-Vergleich

s sollte auch erwähnt werden, daß die Berechnung in der Hardware fast unabhängig n der Sun-Workstation abläuft: Nach der Synthese findet die ganze Berechnung r Gewichtsverteilung auf der EVC1-Karte statt, so daß der Wirt für parallele, nabhängige Aufgaben frei ist.

\.4 Programmierung in höheren Sprachen

)ieser Abschnitt diskutiert, wie die in Abschnitt A.2 vorgestellte Schaltung automa-ch aus einem Algorithmus in einer höheren Programmiersprache gewonnen werden nnte.

). es eine Pipeline-Schaltung ist, wäre eine Synthese aus Vektor-Code — vergleiche Abschnitt 4.3.2 — möglich. Abbildung A.2 zeigt ein Gewichtsverteilungs-Programm Vektor-Code, in dem alle Variablen Daten-Vektoren repräsentieren.

```
(1) I := 1;                (* erzeugt Vektor 1, 1, 1, ...  *)
(2) I := ADD-SCAN(I);      (* erzeugt Vektor 1, 2, 3, ...  *)
(3) C := MAT-MULT(I,G);    (* Matrizenmultipl. mod 2 mit G *)
(4) W := WEIGHT(C);        (* berechnet Gewicht von C      *)
(5) INCR-RAM(COUNT,W);     (* erhoeht Element W in COUNT   *)
```

Abbildung A.2: Vektor-Code für Gewichtsverteilung

-i könnten mehrstufige Implementierungen der Operatoren MAT-MULT und IGHT automatisch durch bekannte Synthese-Methoden erhalten werden. Jedoch e der Operator INCR-RAM zu der daten-parallelen Sprache in [Guc95] hinzu-t werden. Er benötigt ein lokales Steuerwerk, um in einem Pipeline-Takt den l lesen und schreiben zu können. Die optimierte Speicherallokation (Zähler teil-e on-chip, teilweise extern) mit dem Mechanismus zur Ausnahmebehandlung : jedoch kaum automatisch erreicht werden. Dasselbe gilt für die manuell er-en Zeitbedingungen.

nächste Schritt ist die Erzeugung der Schaltung aus einem sequentiellen Pro-m, das keine speziellen Vektor-Anweisungen benötigt. Abbildung A.3 zeigt ein

- Gewichtsverteilungs-Programm in einer imperativen, sequentiellen Programmiersprache.

```
(1) FOR W:=0 TO N DO
(2)     COUNT[W] := 0;              (* initialisiert Gewichtszaehler *)
(3) END
(4)
(5) FOR I:=0 TO (2^K)-1 DO
(6)     C := MAT-MULT(I,G);         (* Matrizenmultipl. mod 2 mit G  *)
(7)     W := WEIGHT(C);             (* berechnet Gewicht von C       *)
(8)     COUNT[W] := COUNT[W] + 1;   (* erhoeht Gewichtszaehler       *)
(9) END
```

Abbildung A.3: Gewichtsverteilungs-Programm in sequentieller Sprache

In Kapitel 5 dieser Arbeit wurde ein Verfahren zur Pipeline-Synthese aus sequentiellen Sprachen vorgestellt. Es kann jedoch auf das Programm in Abbildung A.3 nicht angewendet werden, da die FOR-Schleife in den Zeilen 5 bis 9 nicht vektorisierbar ist. Denn in Zeile 8 wird schreibend auf ein allgemeines Feld zugegriffen, wodurch eine schleifengetragene Datenabhängigkeit entsteht, die nicht durch die in Kapitel 5 vorgestellten Methoden behandelt werden kann. Die Zeilen 6 und 7 können also überlappt in einer Pipeline ausgeführt werden, aber die Iterationen der Zeile 8 dürfen nicht überlappen. (Dies ist die in Abschnitt A.2 beschriebene Begrenzung des Gesamtdurchsatzes.) Es wäre also die in Abschnitt 9.1.2 erläuterte Erweiterung zur Behandlung von Speicherzugriffen in Pipelines notwendig. Außerdem wird für die Funktionen MAT-MULT und WEIGHT die Erweiterung für spezielle kombinatorische Funktionen (Abschnitt 9.1.1) benötigt. Aber auch dann muß die Schaltung noch weiter optimiert werden, um die Geschwindigkeit des manuellen Entwurfs zu erreichen.

A.5 Zusammenfassung

Unsere Experimente zeigen, daß auch sehr kleine SP-Rechner für rechenintensive wissenschaftliche Probleme nützlich sind. Nur viel teurere Parallelrechner können für die Berechnung der Gewichtsverteilung die Leistung unserer rekonfigurierbaren Implementierung erreichen. Deshalb erweitert die Verwendung des EVC1 die Klasse der Codes, für die eine Qualitätsbewertung aufgrund von Gewichtsverteilungen machbar ist.

Diese Anwendung zeigt auch, daß sogar Hardware-Entwürfe, die nur einmal "benutzt" werden, sinnvoll sind. Dies ist eine neue, besondere Eigenschaft rekonfigurierbarer Systeme, die zeigt, daß SP-Rechner für Probleme eingesetzt werden können, für die ein ASIC-Entwurf überhaupt nicht in Frage käme. Damit diese Anwendun-

gen verbreitet eingesetzt werden, benötigt der Wissenschaftler jedoch Werkzeuge, um SP-Rechner ohne Kenntnisse im Hardware-Entwurf programmieren zu können.

Anhang B

Ganzzahliges lineares Programm zur HW/SW-Partitionierung

Dieser Anhang formalisiert das Problem der Hardware/Software-Partitionierung zur Programmbeschleunigung mit programmspezifischen Koprozessoren als ganzzahliges lineares Programm (engl. integer linear program, ILP), so daß die Simplex- und "Branch-and-Bound"-Optimierungsalgorithmen zur Bestimmung der Partitionierung eingesetzt werden können. Im zugrundeliegenden Maschinenmodell können die Koprozessoren nicht direkt auf den Hauptspeicher des Wirts zugreifen, und die Kommunikation zwischen Wirt und Koprozessor-Karte ist langsam. Deshalb sind detaillierte Kenntnisse des Datenflusses nötig, um die Kommunikationskosten zu minimieren. Das Verfahren bestimmt gleichzeitig, welche Programmteile in Hardware implementiert werden sollten und welche Daten dann kopiert werden müssen.

Für den MODULA PIPELINE COMPILER ist dieser Lösungsansatz jedoch nicht sinnvoll, da er von kleinen, sequentiellen Koprozessoren ohne Pipelining ausgeht und dynamische Rekonfigurierung nicht berücksichtigt. Wie in Abschnitt 4.2 erwähnt, ist er eher für das Hardware/Software-Codesign eingebetteter Systeme geeignet.

Im folgenden Abschnitt werden zunächst die Rahmenbedingungen der Partitionierung erläutert. Danach beschreibt Abschnitt B.2 die nötigen Vorverarbeitungsschritte, bevor das ILP vorgestellt wird. Schließlich wird das für ein Beispielprogramm erreichte Ergebnis erläutert.

Dieses Verfahren wurde bereits in [Wei95] veröffentlicht.

B.1 Hardware/Software-Codesign

Im software-orientierten Hardware/Software-Codesign wird ein System beschleunigt, indem zeitkritische Teile eines Programms in Hardware ausgeführt werden. Dazu müssen die geeigneten Programmteile durch ein *Partitionierungsverfahren* bestimmt werden.

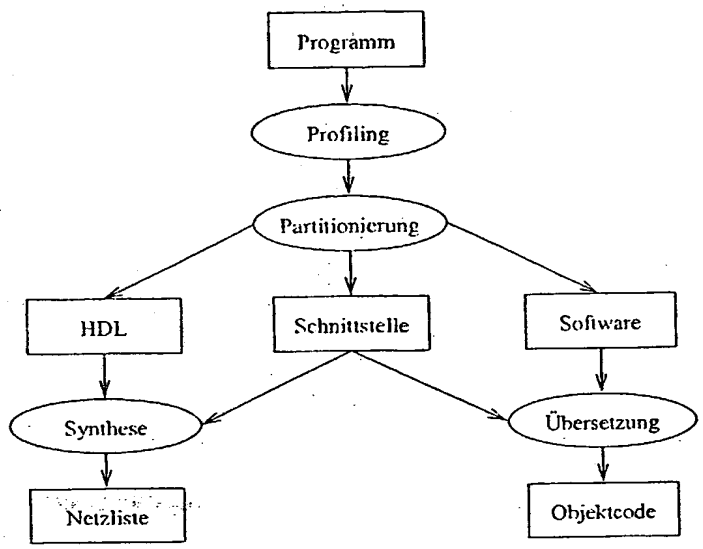


Abbildung B.1: Hardware/Software-Codesign

Abbildung B.1 zeigt den allgemeinen Aufbau eines Codesign-Systems: Ein Profiler bestimmt die kritischen Regionen des Eingabe-Programms und die durchschnittliche Größe der Datenstrukturen, die in diesen Regionen verwendet werden. Diese Informationen werden dann benutzt, um eine realisierbare Hardware/Software-Partitionierung zu finden, die die geschätzte erreichbare Beschleunigung maximiert. Aus den Hardware-Regionen wird dann durch High-Level-Synthese eine Koprozessor-Schaltung erzeugt. Der Software-Teil wird von einem konventionellen Übersetzer verarbeitet, und die Schnittstelle bestimmt die Synchronisation und Kommunikation zwischen Hardware und Software.

Unser Berechnungsmodell geht von einer Koprozessor-Karte mit FPGAs und lokalem Speicher aus, die mit einem Wirt über den System-Bus verbunden ist. Wirt und Karte können über den Bus kommunizieren, aber die Karte kann nicht direkt auf den Hauptspeicher zugreifen, und die Kommunikation ist langsam. Unser Ziel ist es, diejenigen Programmregionen zu wählen, die in Hardware die größte Beschleunigung ergeben. Die Berechnung dieser Partitionierung sollte effizient sein und die Kommunikationskosten zwischen Wirt und Koprozessor-Karte berücksichtigen. Durch den eingeschränkten Zugriff auf den Speicher des Wirts gelten für die Regionen, die in Hardware implementiert werden können, einige Einschränkungen. Außerdem müssen die in den Hardware-Regionen verwendeten Daten explizit zu der Karte kopiert werden. (Wir allokieren FPGA-Register für skalare Variablen und lokalen Speicher für Felder.)

Andererseits können der Wirt und die Koprozessor-Karte gleichzeitig arbeiten, da keine Speicherzugriffskonflikte möglich sind. Die kommerziell erhältliche EVC1-Karte (vgl. Abschnitt 7.1.1) ist ein Beispiel einer Karte, die dieses Berechnungsmodell implementiert.

B.2 Vorverarbeitung

B.2.1 Kandidaten-Auswahl

Einer der wichtigsten Punkte bei der Hardware/Software-Partitionierung ist die Auswahl adäquater Kandidaten, die für eine Hardware-Implementierung betrachtet werden sollten. Man muß zwischen der Granularität der Kandidaten und der Effizienz des Optimierungsprozesses abwägen. Je kleiner die Kandidaten sind, desto größer ist die Zahl der möglichen Partitionierungen. Aber die große Zahl der Lösungen kann die Berechnung des Optimums unmöglich machen. Größere Kandidaten beschränken die Zahl der möglichen Partitionierungen, ermöglichen aber die Berechnung einer exakten Lösung des Optimierungsproblems.

Ernst et al. [EHB93] betrachten jede C-Anweisung als Kandidaten. So werden auch Programm-Regionen, die wahrscheinlich keine Beschleunigung ergeben, betrachtet. Da dies eine große Zahl möglicher Lösungen ergibt, wird eine heuristische Methode (Simulated Annealing) verwendet, um eine Partitionierung zu bestimmen.

Dagegen betrachten Jantsch et al. [JEO+94b, JEO+94a] nur vorausgewählte Kandidaten, die in Hardware implementiert werden können und für die eine Beschleunigung erwartet wird. So kann die optimale Lösung des Partitionierungs-Problems durch dynamisches Programmieren gefunden werden.

Wir übernehmen Jantschs Definition eines Kandidaten [JEO+94b, S. 97]: Eine Programm-Region ist ein Kandidat

- wenn sie keine Gleitkomma-Operationen und keine Aufrufe externer Bibliotheks- oder Betriebssystemfunktionen enthält
- und
  - wenn sie eine innere Schleife oder Funktion ist
  - oder nur Schleifen oder Aufrufe von Funktionen enthält, die selbst Kandidaten sind.

Abbildung B.2 zeigt die Struktur eines Beispiel-Algorithmus. Er erzeugt ein Stereogramm (genaugenommen ein SIRDS, Single Image Random Dot Stereogram) einer Graustufen-Datei, die eine dreidimensionale Szene beschreibt [TIW94]. Die Regionen in den eckigen Klammern wurden weggelassen, damit die Abbildung klarer wird. Die nach der obigen Definition bestimmten Kandidaten-Regionen sind mit `H1`, `H2` und `H3` markiert.

```
for (y=0; y <= maxY; y++) {
   for (x=0; x <= maxX; x++)                                    /* H1 */
      same[x] = x;                                              /* H1 */
   for (x=0; x <= maxX; x++) {
      < non-candidate code defining left, right >
      if (0 <= left && right <= maxX) {
         l = same[left];                                        /* S1 */
         < candidate inner loop using l, left, right, same;     /* H2 */
                               defining same, left, right >     /* H2 */
         same[left] = right;                                    /* S2 */
      }
   }
      < candidate inner loop using maxX, same; defining pix > /* H3 */
      < non-candidate code using pix >
}
```

Abbildung B.2: Struktur des Beispiel-Programms

Wir passen die oben vorgestellte Methode folgendermaßen den Anforderungen unseres Berechnungsmodells an:

- Zuweisungen können zu einer Region hinzugefügt werden, wenn sie Variablen benützen oder definieren, die auch in der Region vorkommen. Dies kann die Menge der zu kopierenden Daten beträchtlich reduzieren. In dem Beispiel der Abbildung B.2 erhalten wir zusätzliche Kandidaten, indem die Zuweisung S1 oder S2 oder beide zu H2 hinzugefügt werden, da same in H2 und in beiden Zuweisungen vorkommt.

- Da es keinen direkten Speicherzugriff gibt, müssen alle Referenzen in einem Kandidaten zur Übersetzungszeit bekannt sein, damit die Daten vor der Ausführung des Kandidaten zur Karte kopiert werden können. Dies verhindert die Verwendung von Zeigern in den Kandidaten und bedeutet, daß Felder im allgemeinen komplett kopiert werden müssen.

- Jantsch et al. betrachten nur Kandidaten, die lokal eine Beschleunigung bewirken. Wir betrachten aber auch Kandidaten, die ein Programm nur in Kombination mit anderen Regionen beschleunigen, in dem sie global die Kommunikationskosten reduzieren.

B.2.2 Datenfluß-Analyse

Aus der Theorie des Übersetzerbaus ist bekannt, daß die Kommunikationsanforderungen von Programmteilen durch Datenfluß-Analyse bestimmt werden

können. Diese Methode kann auch benutzt werden, um die durch eine gewählte Hardware/Software-Partitionierung implizierte Kommunikation zu bestimmen. Wir müssen globale *Definiert-Benutzt-Ketten* [ASU86, S. 632] der Variablen in den Kandidaten bestimmen. Sie stellen eine Beziehung zwischen der Definition einer Variable (z. B. eine Zuweisung an die Variable) und den Stellen, an denen dieser Wert benutzt wird (d. h. Auswertungen der Variable ohne dazwischen liegende erneute Definition), her. Aus den Ketten ergeben sich folgende Bedingungen für eine in Hardware implementierte Region H:

- Falls Variable v in Region H definiert und in Software benutzt wird, muß v *nach* der Ausführung von H *zum Wirt* kopiert werden.
- Falls Variable v in Software definiert und in H benutzt wird, muß v *vor* der Ausführung von H *zur Koprozessor-Karte* kopiert werden.

B.3 Lineares Programm zur Partitionierung

Im folgenden nehmen wir an, daß die Kandidaten-Regionen und die Definiert-Benutzt-Ketten nach Abschnitt B.2 berechnet wurden. Wir werden nun das folgende Partitionierungs-Problem lösen:

> Finde die Teilmenge der Kandidaten-Regionen, die in Hardware implementiert werden müssen, damit die größte geschätzte Beschleunigung für das ganze Programm erreicht wird. Die Schätzung muß die Hardware-Beschleunigung, die implizierten Kommunikationskosten und die Begrenzung der zur Verfügung stehenden Hardware berücksichtigen.

Da zwischen allen Kandidaten eine Definiert-Benutzt-Relation bestehen kann, kann die Entscheidung, eine Region in Hardware zu implementieren, die notwendigen Kopier-Operationen aller anderen Kandidaten beeinflussen. Folglich kann dieses Problem nicht zerlegt werden; wir können ein globales Optimum nur finden, wenn wir alle Kandidaten gleichzeitig betrachten. Wegen der beschränkten Zahl der Kandidaten ist eine exakte Lösung des Problems möglich, indem es als *0, 1 ganzzahliges lineares Programm* [Len90, Kap. 4.6] repräsentiert und mit Standard-Methoden gelöst wird.

Ein 0, 1 ganzzahliges lineares Programm wird durch eine reelle $m \times n$ Matrix A, einen reellen m-Vektor b und einen reellen n-Vektor c definiert. Alle Vektoren $x \in \{0,1\}^n$, die die linearen Ungleichungen (Bedingungen) $A \cdot x \leq b$ und $x \geq 0$ erfüllen und die lineare Kostenfunktion $c^T \cdot x$ minimieren, sind Lösungen.

B.3.1 Notation

Tabelle B.1 definiert die erforderliche Notation. Die Definiert-Benutzt-Ketten werden durch das Prädikat $DU(i,j,v)$ für $i,j \in I_0$, $v \in V$ repräsentiert:

I	Menge der Kandidaten-Indizes
$\{H_i\}_{i \in I}$	Menge der Kandidaten-Regionen
$I_0 = I \cup \{0\}$	Erweiterte Menge der Kandidaten-Indizes
$\{H_i\}_{i \in I_0}$	Menge der Kandidaten-Regionen einschl. Pseudo-Region
H_0	Pseudo-Region (Code, der immer in Software implementiert wird)
$call(i)$	geschätzte Anzahl der Aufrufe von H_i
$HWtime(i)$	geschätzte Hardware-Ausführungszeit für H_i
$SWtime(i)$	geschätzte Software-Ausführungszeit für H_i
$HWarea(i)$	geschätzter Hardware-Bedarf (Zahl der Logik-Blöcke) für H_i
$HWres$	verfügbare Hardware-Ressourcen (Zahl der Logik-Blöcke)
V	Menge der in Kandidaten benutzten Variablen
$size(v)$	Größe von $v \in V$ in Bytes (bei Feldern geschätzte Durchschnittsgröße)
$Ttime$	Dauer des Kopierens eines Byte zwischen Wirt und Karte
$v \triangleleft H_i$	$v \in V$ kommt in H_i vor

Tabelle B.1: Notation

$DU(i, j, v) \Leftrightarrow v$ wird in H_i definiert und in H_j benutzt.

Um alle Abhängigkeiten einheitlich behandeln zu können, repräsentiert eine Pseudo-Region H_0 die Programmteile, die immer in Software implementiert werden. Und für überlappende Regionen müssen Abhängigkeiten innerhalb einer Region aus DU entfernt werden.

Das Partitionierungs-Problem wird durch die folgenden binären Variablen repräsentiert:

$$\forall i \in I_0 : x_i = \begin{cases} 1, & \text{falls } H_i \text{ in Hardware implementiert ist} \\ 0, & \text{sonst} \end{cases}$$

$$\forall i \in I, v \subset V, v \triangleleft H_i : in_{i,v} = \begin{cases} 1, & \text{falls } v \text{ vor der Ausführung von } H_i \\ & \text{zur Karte kopiert werden muß} \\ 0, & \text{sonst} \end{cases}$$

$$\forall i \in I, v \subset V, v \triangleleft H_i : out_{i,v} = \begin{cases} 1, & \text{falls } v \text{ nach der Ausführung von } H_i \\ & \text{zum Wirt kopiert werden muß} \\ 0, & \text{sonst} \end{cases}$$

B.3.2 Kostenfunktion

Die Lösung des ganzzahligen linearen Programms minimiert die Gesamtlaufzeit des Programms, die durch die folgende *Kostenfunktion* $C = C_1 + C_2$ repräsentiert wird, wobei

$$C_1 = \sum_{i \in I} [HWtime(i) - SWtime(i)] \cdot call(i) \cdot x_i \tag{B.1}$$

$$C_2 = \sum_{i \in I, v \in V, v \triangleleft H_i} [in_{i,v} + out_{i,v}] \cdot Cnum(i,v) \cdot size(v) \cdot Ttime \qquad \text{(B.2)}$$

C_1 repräsentiert die Ausführungszeit der Kandidaten. Man beachte, daß die Ausführungszeit der in Software implementierten Regionen nicht gezählt wird, da wir diejenige Partitionierung finden müssen, die die Ausführungszeit *relativ* zur reinen Software-Lösung (für die $C_1 = 0$ gilt) minimiert. C_2 repräsentiert die Kommunikationskosten. Da Konstanten nur einmal kopiert werden müssen, definieren wir für Konstanten $Cnum(i,v) = 1$ und andernfalls $Cnum(i,v) = call(i)$. Für die reine Software-Lösung ist $C_2 = 0$.

B.3.3 Bedingungen

Die folgenden Bedingungen werden zur Definition zulässiger Lösungen benötigt:
Der Software-Teil kann niemals in Hardware implementiert werden:

$$x_0 = 0 \qquad \text{(B.3)}$$

Diese Bedingung für die Pseudo-Region H_0 wird aus den obengenannten technischen Gründen benötigt.

Aus einer Menge von überlappenden Kandidaten kann nur einer in Hardware implementiert werden:

$$\forall i \in I : \sum_{j \in I, H_i \cap H_j \neq \emptyset} x_j \leq 1 \qquad \text{(B.4)}$$

In Abbildung B.2 trifft diese Bedingung auf H2 und seine Erweiterungen zu.

Die beiden folgenden Mengen von Bedingungen repräsentieren die Abhängigkeiten aus Abschnitt B.2.2.

Falls $DU(i,j,v)$, H_i in Hardware und H_j in Software, dann muß v nach der Ausführung von H_i zum Wirt kopiert werden:

$$\forall i,j \in I_0, v \in V, DU(i,j,v), H_i \not\supseteq H_j : x_i - \sum_{k \in I_0, H_k \supseteq H_j} x_k \leq out_{i,v} \qquad \text{(B.5)}$$

Falls $DU(i,j,v)$, H_j in Hardware und H_i in Software, dann muß v vor der Ausführung von H_j zur Koprozessor-Karte kopiert werden:

$$\forall i,j \in I_0, v \in V, DU(i,j,v), H_j \not\supseteq H_i : x_j - \sum_{k \in I_0, H_k \supseteq H_i} x_k \leq in_{j,v} \qquad \text{(B.6)}$$

Man beachte, daß ein Kandidat als in Hardware implementiert gilt, wenn dies für ihn selbst oder für eine Region, die ihn enthält, der Fall ist. (Deshalb stehen die Summen in den obigen Ungleichungen.)

Schließlich müssen die ausgewählten Kandidaten in die verfügbaren Hardware-Ressourcen passen:

$$\sum_{i \in I} HWarea(i) \cdot x_i \leq HWres \qquad \text{(B.7)}$$

B.4 Ergebnisse

Wir haben die Methode auf ein Beispiel-Programm angewendet, das den Algorithmus aus Abbildung B.2 enthält. Zur Lösung des ganzzahligen linearen Programms benutzten wir den "mixed IP-solver" [Ber92]. Die Methode bestimmte, daß die Regionen `H1`, `H2` (erweitert um `S1` und `S2`) und `H3` in Hardware implementiert werden sollen. Im resultierenden System wird das Feld `same`, das Zwischenergebnisse enthält, nur in Hardware-Regionen verwendet und muß deshalb nie kopiert werden — es muß im Speicher des Wirts nicht einmal allokiert werden. Kopier-Anweisungen werden nur für `maxX` (zu den Regionen `H1` und `H3`), `left`, `right` (zur erweiterten Region `H2`) und `pix` (von der Region `H3`) eingefügt. Dieses Beispiel zeigt, daß die Betrachtung genauer Datenfluß-Informationen in der Partitionierung einen bedeutenden Einfluß auf die Gesamtqualität der resultierenden Hardware/Software-Lösung hat.

B.5 Zusammenfassung

Wir präsentierten ein ganzzahliges lineares Programm für die Hardware/Software-Partitionierung im software-orientierten Codesign, das erstmalig detaillierte Datenfluß-Informationen berücksichtigt. Damit können die Regionen, die in Hardware implementiert werden sollten, zusammen mit den implizierten Kopier-Operationen effizient bestimmt werden. Wir haben den Einfluß der Methode auf die Auswahl der Hardware-Regionen demonstriert. Die Methode kann auch für große Programme eingesetzt werden, da die Zahl der Kandidaten in den meisten Fällen begrenzt ist. Obwohl ganzzahlige lineare Programme im allgemeinen NP-hart sind, können praktische Probleme moderater Größe effizient mit Standard-Algorithmen gelöst werden.